



## **Information Technology**

### **Information Technology and Communications (ITCom)**

#### **FINAL PROJECT**

#### **EVALUATION OF POSTGRESQL REPLICATION AND LOAD BALANCING IMPLEMENTATIONS**

**Author: Mikko Partio  
Instructor: Juhani Rajamäki  
Supervisor: Janne Korhonen**

**Approved: March 22, 2007**

**Juhani Rajamäki  
Senior Lecturer**

## PREFACE

This final project has given me valuable experience in the field of high availability databases, and I have come to realise how important it is to ensure data integrity at all times.

The whole process of writing this final project was very straining: with lots of background material and two separate goals to accomplish, the amount of work was enormous, and it is safe to say that without help I would have never made it.

I would like to thank the Finnish Meteorological Institute for providing me with this final project subject, and I would especially like to thank my supervisor Janne Korhonen for making all the practical arrangements and providing assistance when necessary, my project supervisor Juhani Rajamäki for making sure that the contents of the study are sound, and the kind folk at the LT2 project for providing me information regarding the subject. The biggest thanks go to my wife Tiina, who has patiently endured me while I spent endless hours tapping my laptop. Without you this project would have never finished.

Mikko Partio

Helsinki, March 22, 2007

## ABSTRACT

Name: Mikko Partio	
Title: Evaluation of PostgreSQL Replication and Load Balancing Implementations	
Date: March 22, 2007	Number of pages: 59
Department: Computer engineering	Study Programme: Information Technology and Communication
Supervisor: Janne Korhonen, BcS, Group Manager, FMI Instructor: Juhani Rajamäki, Senior Lecturer	
<p>In this final project the high availability options for PostgreSQL database management system were explored and evaluated. The primary objective of the project was to find a reliable replication system and implement it to a production environment. The secondary objective was to explore different load balancing methods and compare their performance.</p> <p>The potential replication methods were thoroughly examined, and the most promising was implemented to a database system gathering weather information in Lithuania. The different load balancing methods were tested performance wise with different load scenarios and the results were analysed.</p> <p>As a result for this project a functioning PostgreSQL database replication system was built to the Lithuanian Hydrometeorological Service's headquarters, and definite guidelines for future load balancing needs were produced.</p> <p>This study includes the actual implementation of a replication system to a demanding production environment, but only guidelines for building a load balancing system to the same production environment.</p>	
Keywords: PostgreSQL, high availability, replication, load balancing, database	

## TIIVISTELMÄ

Tekijä: Mikko Partio	
Työn nimi: PostgreSQL-tietokannan replikaatio ja kuormantasaus	
Päivämäärä: 22.3.2007	Sivumäärä: 59
Koulutusohjelma: Tietotekniikka	Suuntautumisvaihtoehto: Information Technology and Communication
Työn ohjaaja: ryhmäpäällikkö Janne Korhonen Työn valvoja: lehtori Juhani Rajamäki	
<p>Tämä insinöörityö tehtiin sekä Ilmatieteen laitokselle että Ilmatieteen laitoksen Liettuan säähavaintoverkon parantamiseen tähtäävälle LT2-projektille. Työssä selvitettiin PostgreSQL- tietokannan replikaatio- ja kuormantasausmahdollisuudet vaativia tuotantoympäristöjä ajatellen.</p> <p>Työ aloitettiin perehtymällä replikaation ja kuormantasauksen teoriaan sekä itse PostgreSQL-tietokantaan. Teorian perusteella parhaaksi katsottu replikaatiosovellus asennettiin liettualaiseen säähavaintoja keräävään tietokantajärjestelmään. Paras PostgreSQL-pohjainen kuormantasaussovellus asennettiin Suomessa operoiville tietokantapalvelimille, ja kuormantasauksen tehokkuutta mitattiin erilaisilla suorituskyselytesteillä.</p> <p>Työn lopputuloksena Liettuan ilmatieteen laitokselle rakennettiin toimiva PostgreSQL-tietokannan replikointijärjestelmä. Replikointijärjestelmä sisälsi varsinaisen sovelluksen lisäksi erilaisia ylläpitoon liittyviä ohjelmia, joilla replikaatiota voidaan muokata sekä ohjata. Kuormantasauksen suorituskyselytesteistä saaduista tuloksista muodostettiin ohjeita ja suosituksia tulevaisuuden tarpeita varten.</p>	
Avainsanat: PostgreSQL, tietokanta, replikaatio, kuormantasaus	

## **PREFACE**

## **ABSTRACT**

## **TIIVISTELMÄ**

## **TABLE OF CONTENTS**

## **LIST OF ACRONYMS**

<b>1 INTRODUCTION.....</b>	<b>1</b>
1.1 LHMS and the Project.....	2
1.2 Databases.....	5
1.3 Need for Replication.....	6
1.4 Load Distribution.....	7
<b>2 DATABASES.....</b>	<b>8</b>
2.1 Relational Databases.....	8
2.2 ACID.....	9
2.3 SQL.....	11
2.4 PostgreSQL.....	12
<b>3 REPLICATION.....</b>	<b>13</b>
3.1 Replication Techniques.....	13
3.1.1 Synchronous Replication.....	14
3.1.2 Asynchronous Replication.....	16
3.2 Slony-I.....	18
3.2.1 Concepts.....	18
3.2.2 Functionality.....	20
3.2.3 Advantages and Limitations.....	21
<b>4 LOAD BALANCING.....</b>	<b>22</b>
4.1 Different Load Balancing Methods.....	22
4.1.1 Distributed Load Balancing.....	22
4.1.2 Parallel Load Balancing.....	23
4.2 PgPool-II.....	25
4.2.1 Architecture.....	26
4.2.2 Advantages and Disadvantages.....	28
<b>5 SLONY-I.....</b>	<b>29</b>
5.1 Installation and Configuration.....	30
5.1.1 Prerequisites.....	31
5.1.2 Creating a Cluster.....	33
5.1.3 Adding Relations to Replication.....	34
5.2 Testing Functionality.....	35
5.3 Monitoring and Maintenance.....	36
5.3.1 Switchover and Failover.....	36
5.3.2 Modifying Configuration.....	38
5.3.3 Slony-I Monitoring.....	38
<b>6 PGPOOL-II.....</b>	<b>39</b>
6.1 Installing and Configuring.....	40
6.2 Performance Tests.....	40
6.2.1 Theory Behind Tests.....	41
6.2.2 Master/slave Mode.....	43
6.2.3 Parallel Execution Mode.....	47
6.2.4 Analysis.....	51
<b>7 CONCLUSIONS.....</b>	<b>55</b>

## **REFERENCES**

## LIST OF ACRONYMS

ACID	Atomicity, Consistency, Isolation and Durability
AWS	Automatic Weather Station
DCL	Data Control Language
DDL	Data Definition Language
DML	Data Modification Language
FMI	Finnish Meteorological Institute
GTS	Global Telecommunication System
LHMS	Lithuanian Hydrometeorological Service
MSS	Message Switching System
MVCC	Multi-Version Concurrency Control
PITR	Point In Time Recovery
RAID	Redundant Array of Inexpensive Disks
RDBMS	Relational Database Management System
SQL	Structured Query Language
WAL	Write Ahead Log

## 1 INTRODUCTION

The need for this study rose when the Finnish Meteorological Institute started an EU funded project to upgrade Lithuania's climatological infrastructure. All weather observations from either manned or automatic weather stations were to be recorded to a PostgreSQL database, and the database must be replicated to achieve necessary redundancy. A load balancing system must also be configured and tested for future needs of the Finnish Meteorological Institute.

The purpose of this study is two-folded; firstly and most importantly, to create a simple, working database replication solution for PostgreSQL database management system. This replication system is going to be the core of the totally revised Lithuanian Hydrometeorological Services (LHMS) information technology infrastructure, and it will guarantee that all weather information gathered around Lithuania is secured from data loss. Secondly, the aim is to build a load balancing system and test the reliability and the actual performance of the system. The LT2 project, described in section 1.1, does not expect the load on the databases to be too high in the very near future, but as many new commercial weather products are being outlined at the moment, it is possible that at some point the computing power of a single server is not enough to server all users and therefore a load balancing is needed. The purpose of the second part of this study is to chart the different options for load balancing. The overall objective of this study is to research and implement PostgreSQL database system specific replication and load balancing solutions and evaluate their functionality in standpoint of reliability.

The contents of this study are divided into a theoretical and a practical part. Sections two to four describe the theory behind databases, replication and load balancing, and in sections five and six the theory is put into practice. Section two describes relational databases in general, the SQL standard and introduces the PostgreSQL database management system. Sections three and four describe the theory behind replication and load balancing respectively. The implementation is described in section five which deals with the actual installation and configuration of a replication system, and also introduces some ways to manage and monitor the system. In section six a

load balancing solution is implemented, and performance tests are run in order to evaluate the performance of the load-balancing system. Finally, in section seven all results from previous sections are summarized and conclusions are drawn.

## 1.1 LHMS and the Project

Lithuanian Hydrometeorological Service is an institution responsible for meteorological and hydrological observations and forecasts, administered under the Ministry of Environment of the Republic of Lithuania. It is a member of the World Meteorological Organization since 1992, and offers weather services for Lithuanian institutions and enterprises, participates in international programmes and carries out scientific research. LHMS employs a total of 322 persons. [1] As Lithuania joined the EU, the role of LHMS has become more important due to European Unions concern on environmental issues.

The overall objective of the EU-funded project in Lithuania is to strengthen the institutional capacities of LHMS and relevant institutions in the preparation of implementation of the EU Framework Directive on ambient air quality assessment and management and subsequent Daughter Directives. There have been a total of two projects, the current project is the execution phase for the suggestions by the previous project called LT1 (*Procurement of services for the institutional strengthening in preparation for upgrading of Lithuanian meteorological network*), in which declarations of the meteorological stations, data transmissions and air quality modelling were presented. The current project is called LT2 (*Procurement of services for strengthening of Lithuanian institutional capacities in ambient air quality modelling and forecasting*), and it belongs mainly to the meteorological engineering area, whose results will be utilized in air quality modelling and meteorology. [2, p. 5] In other words, the LT2 project will implement the recommendations suggested by the LT1 project.

The short-term objectives are to have four or five automatic weather stations (AWS) functioning in the year 2006, and eventually as more and more automatic weather stations are implemented they will subside all of the



currently operating 20 manual weather stations. Another important objective is to merge all data previously distributed to different systems to one database management system. The whole data acquisition process will be renewed and a thorough quality control will be added. All systems are carefully documented and the LHMS personnel will be trained to operate the new system. The data flow of the new data acquisition process is shown on Figure 1.

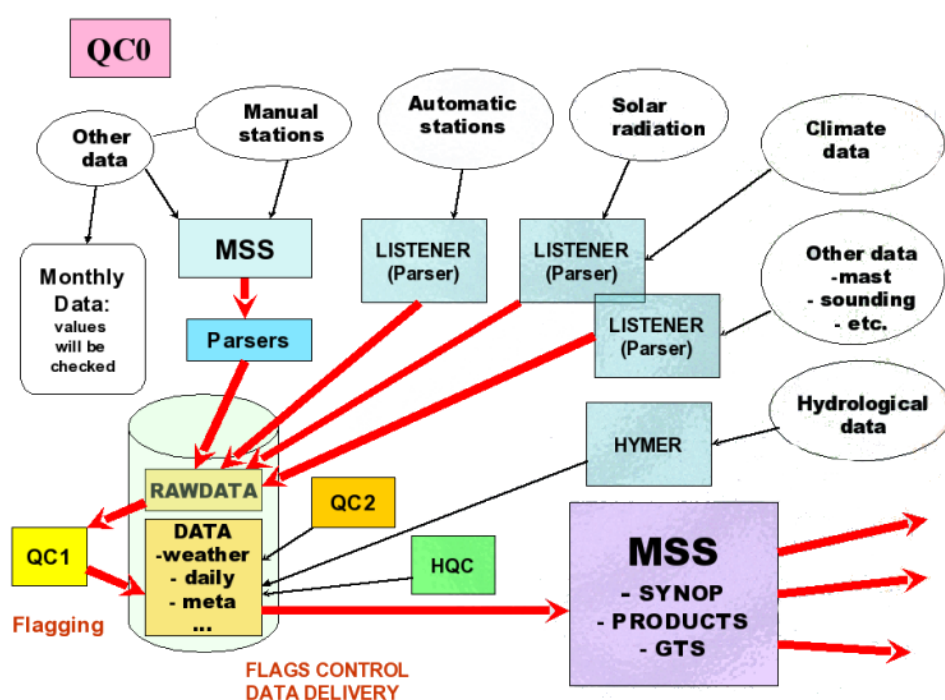


Figure 1. Primary data flow at LHMS [3, p. 2]

The weather data stored in the database is gathered from many different sources. Manual weather stations send their observations every three hours, and their data goes through the Message Switching System (MSS). The MSS is the usual way to pass meteorological information between countries through the Global Telecommunication System (GTS) network, usually with X.25 or tcp/ip protocol. When the data has passed through the MSS, it is parsed to a predefined format and loaded to the database to a rawdata table. Automatic weather stations provide observations every 10 minutes. All observations from automatic stations and other meteorological measuring devices are loaded to the database with a listener/parser technique, where a

simple software listens to incoming data flows, parses the data to the correct format and inserts it to the rawdata table. QC0 means the first stage of quality control, and it is performed by the personnel at the manual weather stations or by the devices and software at the automatic stations.

As observations flow into the rawdata table, the second level of quality control called QC1 starts to filter the data. At all three quality control stages the observations get flagged with an integer value presenting the soundness of the observation. The data is then moved from the rawdata table to other more specific tables. At this point the third level of quality control called QC2 performs horizontal checks with the data, checking all the corresponding observations from different stations and comparing them together. Human Quality Control, HQC, is a method for operators or other personnel to check the values manually. An application for Human Quality Control is being developed by a third party software company in Lithuania.

When the observations have been stored to their corresponding tables, they can be retrieved by users. For each table a view has been created which filters out the values that have been flagged as unreliable, so the users never get to see the neither the actual tables beneath the views nor the bad values they possibly contain. The observations that are sound and get through the quality control system are distributed again through MSS to other countries. In the future commercial weather products are also created from this data.

The core of the whole quality control system is the database hosting the data, and therefore it must be guaranteed that in all cases the data is secured and protected for example from machine malfunctions. One object of this final project is to guarantee the integrity of that database through redundancy: with replication the database can be duplicated so that if one database server is not functioning, a backup server exists and is ready to take over the functions of the original database. This process of replication is explained in section 3. The next sections describes the general concept and the history of modern databases.

## 1.2 Databases

The modern computerized world is controlled by information. The most successful businesses and organizations often have the largest knowledge base of their products and customers. The need of having to archive very large amounts of data and still be able to retrieve it in a very short time has led to the uprising of relational database management systems (RDBMS). An RDBMS is a highly developed piece of software which handles the data archiving and retrieving in the most efficient possible way. Examples of popular database systems are Oracle 10g by Oracle, DB2 by IBM, MsSQL by Microsoft, and the two most notable open source databases systems, MySQL and PostgreSQL. The trend of the recent years has been that for the benefit of the open source movement. According to a recent study, the open source databases are 60% cheaper than their commercial counterparts. The study also cites that although the commercial products have a larger range of features, 80% of the applications use only 30% of the advanced features. [4] This has lead to the situation that traditional database manufacturers like Oracle and IBM are suddenly facing a new threat from an unexpected source.

The history of modern databases is not very long. Relational databases, the databases that are by far the most common type nowadays, were originally developed by two competing projects, the IBM's System R and Ingres started at the University of California, Berkeley. Both of the projects based their work on the mathematical model developed by Edgar F. Codd, an IBM researcher and a mathematician. The System R also developed a intuitive language to access the database, Structured Query Language or SQL, which later on became the industry standard. The flexibility of SQL is one the reasons behind the success of relational databases. [5] As the amount of weather information stored and read daily at LHMS will be quite large, and sophisticated tools are required for the quality control system, the LT2 project decided to use an SQL compliant database called PostgreSQL as the container of the information. PostgreSQL is an open source database management system that was chosen to the LT2 project mainly because of its wide range of features and robust performance.

As databases are a concentration of information, they are often vital to the operation of a business or organization. Therefore, a single database easily becomes a single point of failure. A method called database replication has been created to eliminate this single point of failure. Database replication is explained in general terms in the next section.

### **1.3 Need for Replication**

The concentration of all data to a single archiving system comes with a cost: if a disaster occurs and the database is corrupted or destroyed, all valuable data is lost with it. Studies have shown that only 6% of the companies that suffer a catastrophic data loss survive, 53% never re-opened and other 51% are closed in 2 years time. [6] Therefore it is very important to make sure that the mission-critical data is well protected against hardware failures and other malfunctions that can stop the database from serving users. This means that it has to be guaranteed that if a host serving a database goes down i.e. is unable to respond to queries, a second, standby database exists which then takes over the duties of the first database. This is called failover. The use replication does not remove the necessity for backups, because an ill-placed delete-command on one server gets replicated to the other one(s). Besides the failover property, replication is implemented for various other reasons, such as separating the read and write queries to different servers, distributing the read load for multiple servers or simply transporting data from a central server to subscribing servers. As replication can effectively distribute identical data to multiple servers, load balancing is often accompanied with replication. Load balancers are discussed in the next section.

There are many different ways to achieve replication, with each their own advantages and disadvantages, which will be discussed in more depth in section three. PostgreSQL does not have a built-in support for replication as some other RDBMS' do, but a number of third party developed applications exists, most notable being Slony-I, a trigger-based replication system with support for cascading nodes.

## 1.4 Load Distribution

Generally load distribution (or load balancing) means that processing and communications activity is distributed evenly across a group of nodes in a way that no single server is saturated. In terms of databases, load balancing specifically means distributing the read or write queries across multiple serving databases. The different databases must have identical contents, or the read or written data is not consistent. This dictates the need of replication, and in fact generally a load balancer can not work without an underlying replication system. With load balancing and replication, a high availability database system can be constructed. High availability, in general terms, refers to the fact that a system should be operable 99,999% of the time. This is also known as the five nines principle.

Database load balancing is commonly used in situations where read queries are prevalent over write queries, as is the case for for example most of the Internet based database services. Load balancing can function in two different modes; one option is to split one very expensive query (in terms of disk I/O) to multiple pieces and distribute the pieces to different servers, gather and combine the results and then return the complete result to the user. The other option is that whole queries are sent to different servers depending on some predefined rules. That particular server then processes the whole query and returns the result to the user. The latter way of load balancing, called distributed load balancing, is far more popular due to it's easier implementation and simpler nature. The most popular load balancing solution for PostgreSQL database system, PgPool-II, is capable of doing both type of load balancing methods. PgPool-II is also the load balancer of choice in this final project, and its installation, configuration and performance testing is described in section six.

This introductory section dealt with the very basics of the three essential systems, databases, replication and load balancing. The sections to follow will plunge deeper in to the theory of high availability database systems. The next section describes databases in more detail.

## 2 DATABASES

To understand replication and load balancing, one must know something about the functionality of a database management system. In this section the relational model and its properties are presented, the SQL language is described and the PostgreSQL RDBMS is properly introduced.

### 2.1 Relational Databases

The very first electronic databases were built in the 1960s. They were usually flat files, hierarchically structured trees or based on a networking model which allowed many-to-many relationships between entities. The most common system at that time was called Codasyl, a project backed up by the US Department of Defence. IBM also had its own implementation called IMS. In 1970, an IBM researcher called Edgar F. Codd developed a mathematical model for storing data. The two main points of the model were that (1) the data was independent from the system configuration and (2) a high-level non-procedural language should be used to access the data. At that time Codd's research paper was mainly ignored by the technical community because the hardware was not efficient enough to support an application based on the complex model. Even more resistance came from the employer of Codd, IBM. They had already invested a lot of money to their existing non-relational database system. [7, chapter 6] At that time it seemed that Codd's work was futile, but in fact it was about to change the whole nature of data archiving.

A few years later two projects started to implement a relational database management system based on Codd's model, the IBM's System R and Ingres at UC, Berkeley. Overall the System R did not succeed in convincing the IBM management to use relational databases, but they did develop the high-level language Codd demanded in his model. The language is called Structured English Query Language or SEQUEL (abbreviated later on to SQL). The Ingres project had better success, and backed up by the military they succeeded in developing a fairly accurate implementation of the Codd's original model. [7, chapter 6] Most of the modern database management systems are somehow connected to the original Ingres project.

A relational database consists of relations, and a relation can be depicted as a set of records. A data model is called a schema, and it defines the relations name and the name and the type of each attribute. Additional conditions called integrity constraints can also be defined. For example a unique-condition could be added to one of the attributes. This causes that each tuple (row) in a relation must satisfy the requirements the schema has defined, and the unique-condition defined by the integrity constraint. If a record with a duplicate value for the unique-attribute should be inserted, the transaction would fail. [8, p.11] The advantages of the relational model over the hierarchical model or networking model are unbeatable performance, easy scalability, easy expansion to new hardware technology, flexible usage and the fact that it is very expandable so a solution exists for all types of data needs. [9] The SQL standard defines many other features which are described in more detail in section 2.3. One key feature of a database management system is called ACID. ACID ensures the integrity of the data inside a database by defining a set of rules which must be followed at all times. ACID is introduced and explained in the next section.

## **2.2 ACID**

Atomicity, Consistency, Isolation and Durability (ACID) is the key property of an RDBMS (and the relation model), and nearly all of the database vendors enforce this rule.

Atomicity means that the database guarantees that either all the tasks in a transaction are processed, or none of them are. [8, p. 522] For example if money is transferred from a bank account to another, the bank database guarantees that the money being transferred is first removed from the origin account and then added to the receiving account. If either of these operations fails, the whole transfer fails.

Consistency means that the database is in consistent state before and after a transaction has been processed. [8, pp. 521-522] If the transaction tries to break the logical rules of the database, the integrity constraints, it will be rejected. For example if an integrity constraint holds that a persons age

cannot be more than 120 years, all transactions that try to break this rule are rejected.

Isolation refers to the property that even if multiple concurrent transactions are being processed simultaneously, one transaction cannot see the possible modifications the other transactions are making to the same data. Isolation can also be understood so that the transaction history consists of serialized actions, although for performance issues many transactions can run concurrently. [8, p. 521] This property can be achieved by a process of locking tables, or by a technique called Multi-Version Concurrency Control (MVCC). Locking means that a transaction, a collection of one or more logical operations, can lock the table in such a manner that no other transaction can make modification to it while the lock holds. After the transaction is finished, the transaction releases the lock and allows other transactions to perform. Many different lock types exist, and not all block out other transactions. While the lock technique is simple and easy to use, it does not perform well when the number of concurrent users is high. The other technique, MVCC, takes snapshots of the contents of the database, and only these snapshots are visible to a transaction. Once the transaction is complete, the modifications that were done are applied to the newest copy of the relation and the snapshot is discarded. This means that in any given time multiple different versions of the same data exist. MVCC performs very well even when a large number of users are connected to the database at the same time. The downside is that the discarded copies of snapshots have to be removed by a separate process, and with a large or highly updated dataset removing the discarded copies can be a slow and encumbering procedure.

Durability simply means that if a transaction is successfully committed (i.e. all the actions it has made have been successfully executed at database), the user can rest assured that the transaction cannot be undone, and in the event of system failure the RDBMS will be able to store the database to the state it was before the failure. [8, p.522] Many database management systems maintain a log called Write-Ahead Log which ensures that point in time recovery is possible.



The ACID rule guarantees that the data is safe inside the database, but it does not enable reading or manipulation of the data. For this purpose a language called SQL was invented. SQL is introduced in the next section.

## 2.3 SQL

SQL stands for Structured Query Language. It was originally developed by the IBM's System R project but later on it was standardized by the ISO/ANSI organizations. The latest official standard is called SQL:2003. The SQL standards describe the overall functionality of a relational database, and a set of functions and commands an SQL conforming database should implement. The standard has grown in size significantly over the years, while the SQL-92 standard consists of 1,120 pages, the SQL:2003 standard has spanned to over 3,600 pages. [10] The lengthy definitions and the fact that the standard is often ambiguous and does not describe every detail of the implementations has led to the situation that every RDBMS vendor has implemented the standard their own way. The SQL syntax between vendors is not 100% compatible, but it is more like dialects of a common language.

SQL commands are divided into three different categories: Data Definition Language (DDL), Data Modification Language (DML) and Data Control Language (DCL). DML commands are perhaps the most used ones, they allow the users to manipulate the data, that is to retrieve, insert, update and delete data. DDL commands allow the user to alter the structure of the database, for example altering existing tables and adding or dropping tables is supported by the DDL command subset. DCL commands allow the user to grant or revoke access privileges to relations to other users. [8, pp. 131-132] All three command subsets are supported by all SQL compliant databases.

SQL standard also defines the process of joining two or more tables together, triggers, transaction management, object-oriented features, spatial data and many other features. One notable thing is that although the standard does not define indices, all database implementations support them. SQL language offers many options to modify and work with the data in the database, so it suites very well to quality control system of the LT2 project. As described in section one, the QC1 quality control stage analyses

and modifies the data based on the soundness of the weather observations. The operations are very performance consuming, and with the SQL language the quality control procedures can be done inside the database i.e. no external programming language for the data manipulation has to be used. This improves the performance of the quality control since if the data is manipulated inside the database, it is usually much faster than if the data is first retrieved, modified externally and then inserted back to the database. The next section introduces the PostgreSQL database management system, which is the database of choice for the LT2 project.

## 2.4 PostgreSQL

PostgreSQL is an open source (Object) Relational Database Management System, with a rather long history in the context of databases. It is a spin-off from the original Ingres project. In 1986 the name was changed to Postgres when at that time a new, revolutionary feature called object-orientation was added to core of the system. At this point the database used a Postgres-specific language called Postquel to access data at the database. Somewhere around 1995 SQL-support and many other new features like Multi-Version Concurrency Control were added to the system, and because of the radical improvements the name was also changed to PostgreSQL. [11, p.1] Although the development of PostgreSQL is coordinated by a group of open source developers called the PostgreSQL Global Development Group, anyone can make contributions, as long as the contributions are accepted by the core group of developers.

PostgreSQL is very feature-rich and mature database system, and it implements nearly all SQL-92 standards and many features of the SQL:1999 standard. Some of the most important features of PostgreSQL are superior multi-user concurrent performance through MVCC, point in time recovery with WAL, multiple procedural languages including PL/pgSQL, PL/tcl, PL/perl and PL/python, support to various client APIs and user-created data types. [11, p.1-2] Many third party software contributions have been written by programmers around the world, and they offer additional features such as full text indexing and XML support.

Generally it is easier and more efficient to store a small set of data in a simple file, but when the amount of data becomes larger or the data has to be manipulated or processed, a database is the only option. Relational databases have many advantages over simple flat-file databases, as discussed in this section, and therefore a relation database is also used in the LT2 project. The LT2 project has chosen PostgreSQL out of the multitude of database systems, mainly because its variety of features and because it is open source software. PostgreSQL enables not only the efficient storage and retrieval of the data, but also the necessary manipulation required by the quality control system. This section has described the concept of a database, and the next section introduces and describes in detail the methods, aims and purposes behind database replication.

### **3 REPLICATION**

Replication is one of the key methods to ensure that the data inside a database is safe and serviceable in case for example of a machine malfunction. Replication was briefly introduced in section one, and in this section it is explained in more detail. The replication implementation used in the LT2 project is also introduced and presented in depth.

#### **3.1 Replication Techniques**

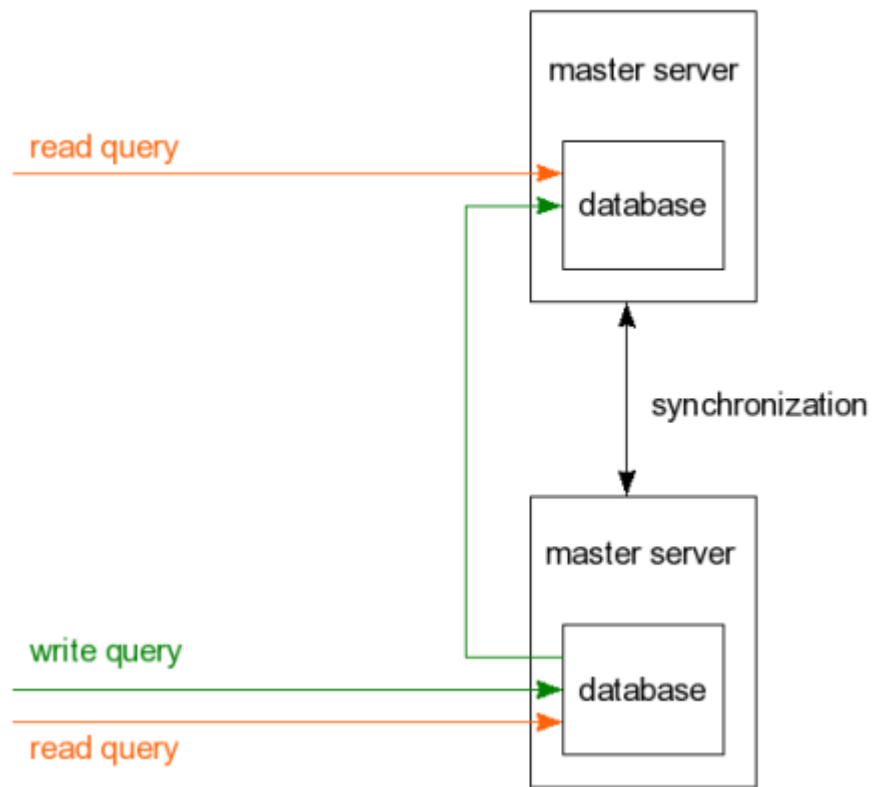
Replication, in the context of databases, means via data distribution two or more database instances ideally have the same contents at all times. Replication can be divided into two different techniques, synchronous and asynchronous replication, and these two methods can both be divided in to master-to-master and master-to-slave subcategories. The difference between synchronous and asynchronous replication is distinct: in synchronous replication the content of the databases that are replicated is always the same, no matter of circumstances. In asynchronous replication, the content of the databases is identical most of the time. Synchronous and asynchronous replication are described in the sections 3.1.1 and 3.1.2, respectively.

Master/master replication means that all replicated databases are able to serve all users individually to the full extent, as opposed to the master/slave method where only one server, called master, can read and write to the database. The other servers, called nodes, can server only read queries. After the master has modified the database it distributes the information to the slaves. The master/master and master-to-slave division is implicit to the synchronous and asynchronous division: master/master method is always used with synchronous replication, and master/slave is nearly always used with asynchronous replication. To facilitate the description, a host running a database will be from now on referred as a node.

All major commercial database vendors offer both synchronous and asynchronous replication. MySQL implements both replication types through different replication engines and clustering solutions. PostgreSQL offers both synchronous and asynchronous replication through free third party applications called PgCluster and Slony-I respectively.

### *3.1.1 Synchronous Replication*

The first method of replication is called synchronous replication. Synchronous replication means that all nodes in a replication cluster have exactly the same content at all times, and all nodes act as “master servers”. A master server is a term meaning a node that can read and write data to a database. All queries that read data from a database can be processed entirely by a single node. All queries that write data to a database are distributed to all the participating replication nodes, and the write operation is permanently committed only when every one the nodes has confirmed the committing of the data. This is problematic because a write operation has to exclusively lock all tables it wishes to modify, and acquiring these locks from different databases on different hosts may take some time. During the time it takes to receive all locks from different nodes it holds on to locks already acquired, therefore preventing any other write operation to that table. If a acquiring a lock from one node takes a long time it might render the other nodes incapable to respond to any queries. [8, p. 750] A diagram of master/master synchronous replication is presented in Figure 2.



*Figure 2. Synchronous master/master replication.*

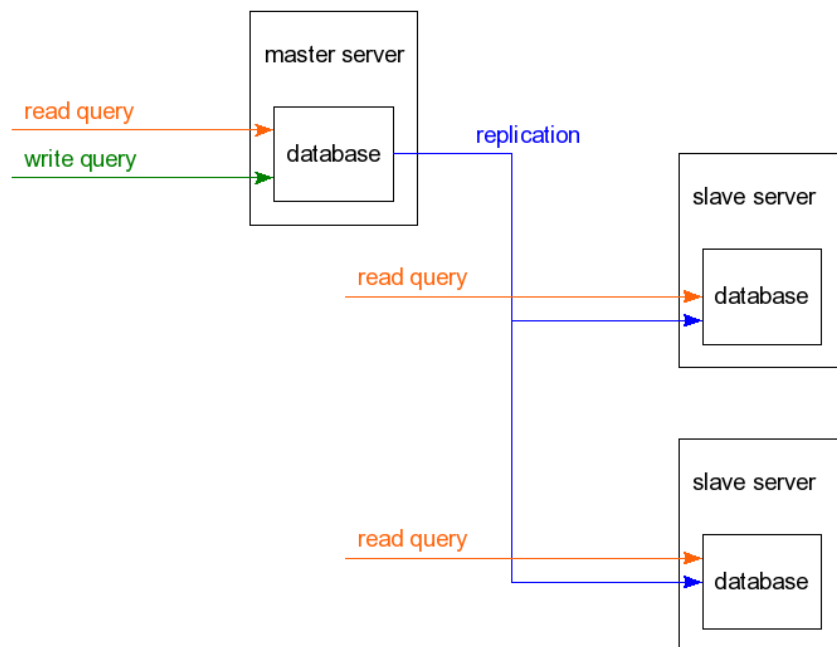
Figure 2 illustrates a situation where a synchronised replication cluster consists of two nodes and one replicated database. When a write query, marked with a green arrow, arrives to one of the nodes it will also be processed by the other node. Read queries, marked with orange arrows, can be processed entirely by the node they first arrive to. Both nodes also make sure that they are synchronised with their counterpart. Because all write queries have to be sent to all nodes, synchronous replication does not suit well to write-oriented databases. Also, when new nodes are added the whole system becomes increasingly complicated, because of the nature of the system.

As explained in the previous section, synchronous replication is always master/master replication. It would be theoretically possible to build a master/slave synchronous replication, but as it would disable the most important feature of synchronous replication, the ability to write data to all replicated databases, it is practically never used. As synchronous replication

can get very complicated and cumbersome, an alternative technique has been invented. This alternative technique is called asynchronous replication, and it is described in the next section.

### 3.1.2 Asynchronous Replication

The second method for replication is called asynchronous replication. It usually consists of a master node, and one or more slave nodes. All write queries are sent to the master node which processes them, and then depending on the technique of choice, sends either the changes made or the whole snapshot of the database contents periodically to the slave nodes. The slave nodes can serve only read queries. During the time which it takes for the master node to replicate the changes to the slave nodes, the two databases do not have the same contents. This replication time can be a few seconds or many hours, depending on the replication implementation, database load and database hardware performance. [8, pp. 751-752] Figure 3 shows an example of asynchronous master/slave replication.



*Figure 3. Asynchronous master/slave replication.*

Figure 3 illustrates an asynchronous master/slave replication cluster with a total of three nodes, and one replicated database. The leftmost node acts as

a master for the cluster. When the master receives a write query, marked with a green arrow, it first writes the data to its database and only after the writing is successful it sends information on the modified data to the slave nodes. This replication data flow is marked with blue arrows. Read queries, marked with orange arrows, can be processed by any of the databases in the replication cluster. A replication cluster is simply a collection of nodes participating to the replication.

When a write operation has been successfully committed at the master node, but not yet distributed to the slave nodes, the database cluster is in an inconsistent state. The situation where the databases have different content clearly breaks the ACID rule, because usually the two or more databases are seen as one outside the replication cluster, and during this particular period of time the result of a read query depends on which node it is executed on. Although asynchronous replication breaks the ACID rule, it is often preferred over synchronous replication because of its easier implementation and simpler architecture. Asynchronous replication suits very well to situation where the minor delay between the shifting of the data does not matter, one example could be a data warehouse where information is stored for archiving purposes.

As mentioned in section 3.1, of the two subcategories of replication, master/master and master/slave, the latter is prevalent in the case of asynchronous replication. Master/master replication is possible in an asynchronous environment, but as it means that all nodes could write data without consulting the other nodes, the master/master method can produce unexpected and even dangerous results. For example if a bank had asynchronous master/master replication, an unwanted situation would be when a user would withdraw money from a bank account, and the first node would write the change of account balance to the database. At this very moment one node would see the account with the money withdrawn, and a second node would still see the original amount. Now if the user would withdraw more money, the worst case scenario for the bank would be that user could withdraw money she did not have, using the account information from the second node. Due to the possibly dangerous inconsistency of data

between nodes with master/master asynchronous replication, it is very rarely used.

Synchronous and asynchronous replication are the two different choices to perform replication, and the actual implementations of these techniques are numerous. In the next section one replication implementation for PostgreSQL is presented.

## 3.2 Slony-I

The LT2 project has after careful consideration decided that asynchronous replication is the preferred method of replication, since a replication delay does not matter as the slave node is not serving any users. After exploring different options and applications for the actual implementation of the replication, one solution clearly rose above others. The application that is used to perform replication in the LT2 project is called Slony-I. Slony-I is a mature master-to-multiple-slaves asynchronous replication system with cascading nodes and PostgreSQL version independence. [12, p. 1] It is written in C and PL/pgSQL -languages, and its functionality is based on triggers attached to replicated tables.

### 3.2.1 Concepts

Slony-I defines a set of concepts which facilitate the understanding of the replication process. For each replicated database, a Slony-I *cluster* is created. A cluster simply identifies the database, and it contains all the information that is required for the replication. Physically a cluster consists of an *origin node*, and one or more *subscriber nodes*. An origin node, or the master node, hosts the primary copy of the database, and is the only node allowed to make updates to it. Subscribing nodes, or slave nodes, listen to the origin node through *slon* processes. Since Slony-I is a cascading replication system, a subscriber can be a forwarding node, which means that the subscriber listens to possible update notifications from the origin node and distributes these updates to other subscribers. Slony-I maintains all information necessary for the replication in a set of relations under a special schema inside the replicated database. The name of this schema consists of an underscore and a cluster name. The smallest replicatable object is called



a *replication set*. A replication set consists of one or more tables or sequences.

Slony-I consists of four entities: slon, slonik, system tables and triggers. As mentioned above, Slony-I creates a special schema called `_clustername` which contains all the necessary information for the replication. Slony-I also adds a trigger to all replicated tables. At the origin node this trigger records all changes made to table and stores them to table `sl_log_1` or `sl_log_2`. At the subscribing node the trigger prevents all modifications to that table. When the changes are recorded to the Slony-I system tables, it is up to slon to distribute them.

Slon is a program that communicates between the replicated nodes and distributes the DDL and DML commands executed at the origin to the subscribers. One slon process is required for each replicated database and for each node in replication cluster. A basic network diagram is shown in Figure 4.

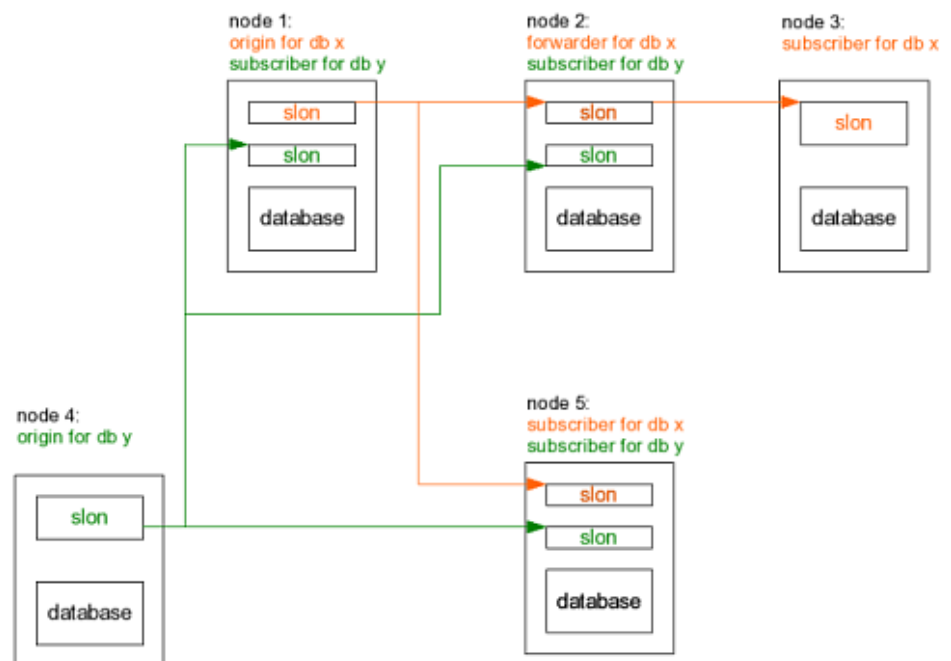


Figure 4. Slony-I network diagram.

In Figure 4 node one acts as an origin for database x and a subscriber for database y. All data concerning database x is marked with orange arrows

and text, and data concerning database y is marked with green arrows and text. Node two acts as a subscriber for database y, and a forwarder for database x, so it is a cascading node. Although node three is a subscriber for database x, it receives all its update from node two, not from the origin of the database. Node four is an origin for database y and finally node five subscribes to both databases. As the figure suggests, Slony-I offers multiple different roles for nodes which makes it a very flexible replication solution.

Slonik is a very simple preprocessor that is used to configure the replication cluster. Slonik has a very limited command syntax, and all Slony-I related commands are executed through Slonik. When for example a new replication cluster is initialized, all the necessary commands are fed through slonik, which then populates the system tables with necessary information, and also creates the triggers to the tables that are replicated. Slonik's functionality is explained in more detail in section five, where the actual replication cluster for the LT2 project is created.

### 3.2.2 Functionality

Essentially it is the slon processes that do the actual distribution of data between nodes. When a replication cluster is initialized and populated with relations (tables or sequences), the slon processes start communication. Note that even though in Figure 4 it is shown that all slon processes reside in the same node as the database, all slon processes can run for example in an external node if required.

The subscribing slon has information on how to contact the originating slon in tables *sl\_path* and *sl\_listen*. When a change occurs to a replicated table, a trigger attached to it will record all information concerning the modification to table *sl\_log\_1* or *sl\_log\_2*. By default the originating slon checks the *sl\_log\_X* table every 10 seconds for possible changes in the databases data. If a change has occurred, slon issues a notification to all subscribers and writes a record of the notification event to table *sl\_event*. When the subscriber sees the notice, it copies the changes from the origin node and after successful application of the changes confirms the origin that the information on the changed data can be purged out from the *sl\_event* table. This information is recorded to *sl\_confirm* and eventually purged out. Slon

must keep on checking the value of sequences regularly, because triggers do not work on sequences. In other words, replicated sequences will create sync events even if no changes have happened. For each replicated sequence a single row is added to `sl_log_X` periodically, which will affect the overall performance if the number of sequences is large.

### *3.2.3 Advantages and Limitations*

Before deciding which replication method and implementation to use, it is important to find out the advantages and disadvantages of the different methods. No one solution can serve all needs, therefore a number of different implementations have been developed over the years. Slony-I has many advantages over competing PostgreSQL replication systems, but it is not by all means a “perfect” solution. On the upside Slony-I offers easy implementation and configuration, solid functionality, techniques based on existing SQL standards and an active user community. Slony-I is also the only asynchronous replication solution for PostgreSQL.

The drawbacks of Slony-I are caused by the technique itself: since PostgreSQL does not allow triggers to be placed on system tables, Slony-I has no way of knowing whether DDL commands have been executed. This means that all such changes have to be executed through `slonik` which can be cumbersome if lots of changes are made. Also, Slony-I can not replicate large objects or certain SQL commands such as `TRUNCATE` since they bypass triggers. All tables that need to be replicated need to have a primary key or other unique index defined, but this is not such a great limitation because a well designed table should have a unique index anyway.

Replication is not easily accomplished. Even the most simple solutions can easily become very complicated when new nodes or databases are added, and because the real value of replication is measured in emergency situations when something has already gone wrong, replication needs a careful design and maintenance from a competent database administrator. A non-working replication can be even more dangerous than having no replication at all, since a faulty replication can lull the administrator into a dangerous wrong sense of security and when something happens, in the worst possible case neither replication nor backups are up to date.

As mentioned, Slony-I is the replication implementation the LT2 project is going to use, and the installation and configuration is described in section five. Slony-I replication can also be combined with a load balancer, in order to achieve the five nines reliability and to effectively distribute the load evenly across a range of nodes. Load balancing is discussed in the next section.

## **4 LOAD BALANCING**

Load balancing is used when the work load of a single server is too much for the server to handle by itself. A load balancer can be used to distribute heavy cpu or disk intensive work to multiple computers, therefore achieving the optimal performance. This section describes the different load balancing methods and the principals behind them. A PostgreSQL based load balancing method called PgPool-II is also introduced. The next section describes the different load balancing architectures.

### **4.1 Different Load Balancing Methods**

To know how load balancing can be accomplished, one must know how a database system with multiple nodes can be implemented. Database systems can generally be divided into two main categories on basis of the amount of resources they share. The two categories are distributed database systems and parallel database systems. A distributed database system consists of multiple independent sites which all contain the same data, or a small subset of it. In a parallel database system usually at least some of the resources are shared. [8, p. 726] Load balancers are divided on the basis of the database system division, there are distributed load balancers and parallel load balancers. Both of these techniques are described in more detail in sections 4.1.1 and 4.1.2. The next section deals with distributed load balancing.

#### ***4.1.1 Distributed Load Balancing***

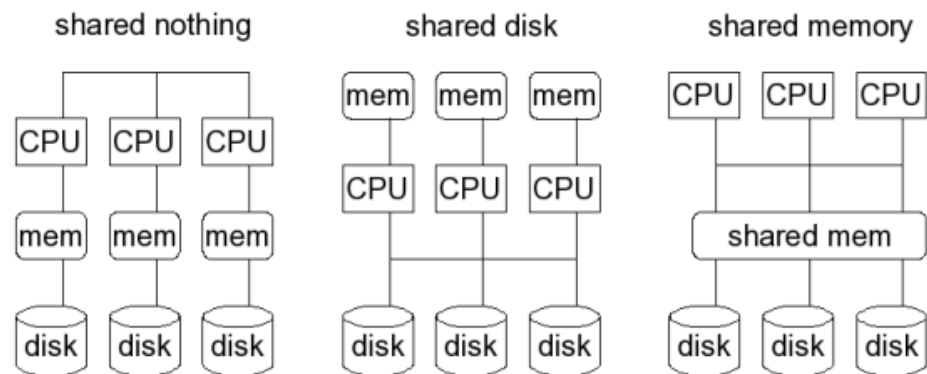
In a distributed database system the data is physically located in several, independent sites. Although these sites function independently, together they form a unified database. The contents of the databases at different sites may be identical, but the contents can also be fragmented horizontally or

vertically. Horizontal fragmentation means that the relation has been divided into fragments which contain a subset of the entire relation. Vertical fragmentation means that the relation has been split so that different fragments contain a subset of the columns of the original relation. [8, pp.726,739-740]. Distributed load balancing means that when a load balancer application receives a query, it will distribute the query in a predefined fashion, for example to the database server with the least load. The database then processes the whole query and returns the results to the load balancer (often referred as a load balancing frontend), which then returns the result to the user. This type of load balancing is useful if the majority of the queries are simple, i.e. do not consist of heavy i/o operations or complicated joins or unions.

In some cases distributed load balancing benefits largely from fragmentation. The problem with fragmentation is the distribution of data; since each node will only have a subset of the data, efficient data distributing between the nodes is a very difficult, if not impossible. PgPool-II, a PostgreSQL based load balancer supports regular distributed load balancing as well as fragmentation-based load balancing, and both methods are implemented in section 6. The next section presents the theory behind parallel load balancing.

#### *4.1.2 Parallel Load Balancing*

A parallel database system is designed to take the performance to the maximum by running various operations simultaneously. Different parallel architectures are divided on basis of the amount of resources they share. The most common architectures are shared nothing, shared memory and shared disk architectures. The different architectures are presented in Figure 5.



*Figure 5. Parallel systems architectures*

In Figure 5 the three different architectures for parallel computing are illustrated. The leftmost architecture called shared nothing-architecture means that all processors (CPUs) in the system have their own memory and disk segments, and all communication between processors goes through an interconnected network. Basically a shared nothing architecture is equals to a distributed database system since the architecture consists of individual processing systems. The architecture in the middle is called a shared disk-architecture, and in that architecture each CPU has its own private memory segment, but the disk space is shared between the processors. In the rightmost architecture, shared memory-architecture, different CPUs are connected together by an interconnection network, and they share all memory and disk. [8, p. 727] The amount of components shared increases from left to right, and usually the more resources are shared the better is the performance of a system.

Parallel load balancing means that when a frontend receives a query, it will inspect that query and split it into different sections if possible. It then distributes the different sections of the query to different database servers which then process their part and return the results to the frontend. The frontend combines the results and returns the final result to the user. Parallel load balancing is useful when majority of the queries are complex, so that they can be split into parts.

In principle parallel load balancing can be faster than distributed load balancing. In reality it is not often so; parallel load balancing introduces new complicating factors that have to be taken into account when designing a

database system. The trend of the recent years has been that for the distributed database systems for two reasons: (1) the prices of traditional servers have decreased considerably so a computing grid of several tens or even hundreds of computers is not such a large investment, and (2) because a distributed system is easily expanded by adding new nodes, as opposed to a parallel supercomputer which after reaching its maximum capacity and performance is usually replaced. In the LT2 project the database system consists of two independent nodes, so distributed load balancing is the technique of choice. The most popular PostgreSQL based load balancing implementation called PgPool-II is presented in the next section.

## **4.2 PgPool-II**

There are two factors that favour distributed load balancing over parallel load balancing when the LT2 projects' database needs are considered. Firstly, retrieving meteorological data from a database rarely includes very complex queries. Secondly, the database system designed for the LT2 project consists of two independent nodes. These two factors combined clearly dictate that the optimal load balancing solution for the LT2 project would be a distributed load balancing system. It should be noted that the actual load balancing system is built to a simulated environment, since the LT2 project does not require load balancing at this point.

PostgreSQL database system does not have a built-in load balancing, but a third party application called PgPool-II has been written by a group of developers to enable load balancing for PostgreSQL databases. PgPool-II is the most auspicious PostgreSQL specific load balancing solution with a variety of features that are described in more detail in the next section. The installation of Pgpool-II to a test environment and the performance testing are described in section 6. In order to support all the different feature, PgPool-II has an architecture designed to be as flexible as possible. The next section describes PgPool-II architecture.

#### 4.2.1 Architecture

PgPool-II is a middleware program for PostgreSQL which offers replication, load balancing and connection pooling. It is a continuation from an original PgPool-I project, and it has many advanced features such as the ability to extended to up to 128 nodes, the ability to execute parallel queries and it also has a web-based managing interface called pgpoolAdmin. [13] As PgPool-II offers parallel query load balancing, it has to have a special architecture to support this feature. PgPool-II architecture is presented in Figure 6.

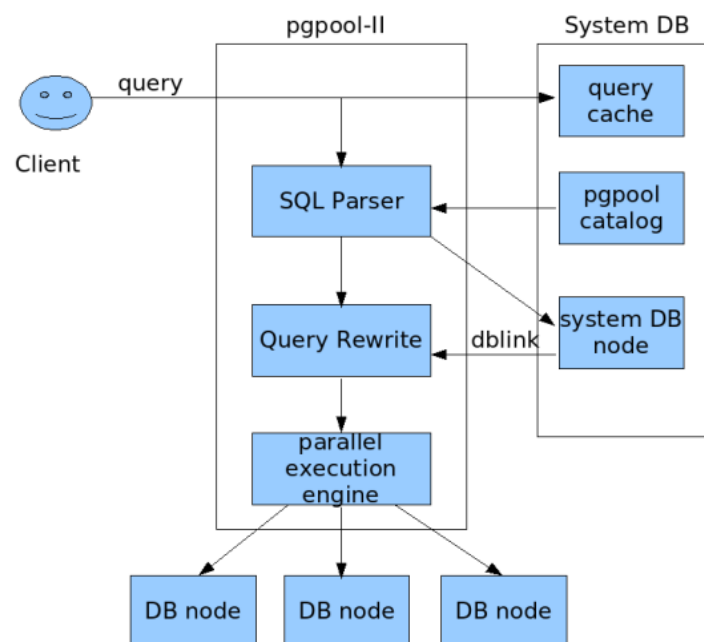


Figure 6. PgPool-II architecture [13]

In Figure 6 the PgPool-II load balancing architecture is illustrated. When a query is first issued by the user, PgPool-II will check if it is already at the cache of the system database. The system database, depicted in the right side of Figure 6, is a regular PostgreSQL database which stores all the necessary information for PgPool-II to operate. If the result exists in the cache, it will be returned to the user immediately. If the query is not cached, PgPool-II examines the possibility of parallel execution. The query is sent to SQL parser, imported from PostgreSQL. If the query is also a read-query the



pgpool catalogues are consulted for the possibility of executing the query in parallel mode. If parallel execution is possible, the query will be rewritten in a proper form so that it can be distributed to the nodes which contain the actual data. If parallel execution is not possible, or it is disabled by the administrator, the query is simply redirected to one of the database servers in the PgPool-II's server pool. The parallel execution mode of PgPool-II is a bit misleading: PgPool-II does not offer real parallel load balancing since PostgreSQL supports only shared nothing - architectures, but PgPool-II supports horizontal fragmentation, which is a kind of a parallel execution mode. PgPool-II has different modes of operation with different features. [14] These features are listed in Table 1.

Table 1. PgPool-II modes and features.

<b>Functions/ Modes</b>	<b>Raw mode</b>	<b>Connection pooling mode</b>	<b>Replicatio n mode</b>	<b>Master/slave mode</b>	<b>Parallel query mode</b>
<b>Connection Pooling</b>	-	X	X	X	X
<b>Replication</b>	-	-	X	-	-
<b>Load Balancing</b>	-	-	X	X	-
<b>Degeneration</b>	-	-	X	X	-
<b>Failover</b>	X	X	-	-	-
<b>Parallel Query</b>	-	-	-	-	X
<b>Required # of servers</b>	one or more	one or more	two or more	two or more	two or more
<b>System DB?</b>	no	no	no	no	yes

In Table 1 the different operational modes are listed in the uppermost row. The different features a mode can have are listed in the leftmost column. An "X" in a column means that the mode supports that feature, and a "-" sign means that the feature is not supported by that mode. Raw mode means that PgPool-II functions only as frontend for clients connecting to the database and it does not do load balancing. Raw mode supports failover, an operation that is used when a nodes is malfunction it has to removed from the server pool. Failover is described in more detail in section 5.1.3. Raw mode does not require the system database. In connection pooling mode PgPool-II spawns multiple processes when at startup, and these processes wait for incoming users. The benefit of this is that when a new user contacts

the database, no new processes has to be spawned, and the response time of the database is smaller. The failover feature is also supported in connection pooling mode.

Replication mode uses PgPool-II's own built-in replication and it also does distributed load balancing between the database servers and supports degeneration of nodes. Degeneration means that when PgPool-II notices that a node participating in replication is malfunctioning, that node can be removed from the server pool automatically. This feature is equivalent to the failover feature of load balancing mode. Master/slave mode is designed to be used with Slony-I replication, and when this mode is enabled PgPool-II directs all write queries to the master node, node number zero, and distributes read queries between all servers in the pool. Connection pooling and degeneration are also supported in this mode. Finally, parallel query mode, the only mode that uses the system DB, does connection pooling and load balancing based on parallel queries. As mentioned earlier, the parallel replication in this context means load balancing based on horizontal fragmentation.

As the contents of the LT2 projects databases are replicated with Slony-I, the obvious choice of PgPool-II mode is master/slave. Besides the master/slave mode, the performance tests executed in section 6 are done also with parallel query mode. The purpose of this is that it makes possible to compare the performance of the load balancer with different load balancing methods. Parallel query mode is really not a feasible option from the LT2 project, as horizontal fragmentation is impossible to implement with Slony-I. The next section summarises PgPool-II's advantages and disadvantages.

#### *4.2.2 Advantages and Disadvantages*

PgPool-II is a versatile and robust load balancer, but it has its limitations. In the replication and master/slave mode only clear text password and trust authentication methods are supported. PgPool-II does not have host based authentication methods similar to PostgreSQL, therefore any access to the PgPool-II frontend must be limited by other means. The underlying PostgreSQL servers accepting PgPool-II connections can have host based authentication. The load balancing modes themselves also introduce

restrictions, for example with master/slave mode load balancing with parallel queries is not possible.

As said, distributed load balancing with PgPool-II requires that the contents of the databases at different nodes are identical. With asynchronous replication this is easy to achieve using for example Slony-I, and in fact PgPool-II has built-in support for Slony-I. PgPool-II also offers a replication solution of its own, but at least for now it is very new and immature, and does not compare with the infallibility of Slony-I.

Databases are very often load balanced since as the amount of data stored in the database increases, and the number of users connecting to the database increases, the database easily gets saturated and can not perform with the necessary swiftness. With load balancing, the saturation is easily avoided, because in addition to dividing the work load to multiple servers, load balancers usually offer connection pooling which also reduces the overhead. Load balancing and replication are often combined, because generally load balancing can not be achieved without replication, since the contents of the load balanced databases have to be identical. Also when a replication system is built, it is moderately easy to add a load balancing system on top of the replication system. In the next section the installation and configuration of a Slony-I replication system is described in great detail, and in the section after that a load balancing solution is built on top of the Slony-I replication system.

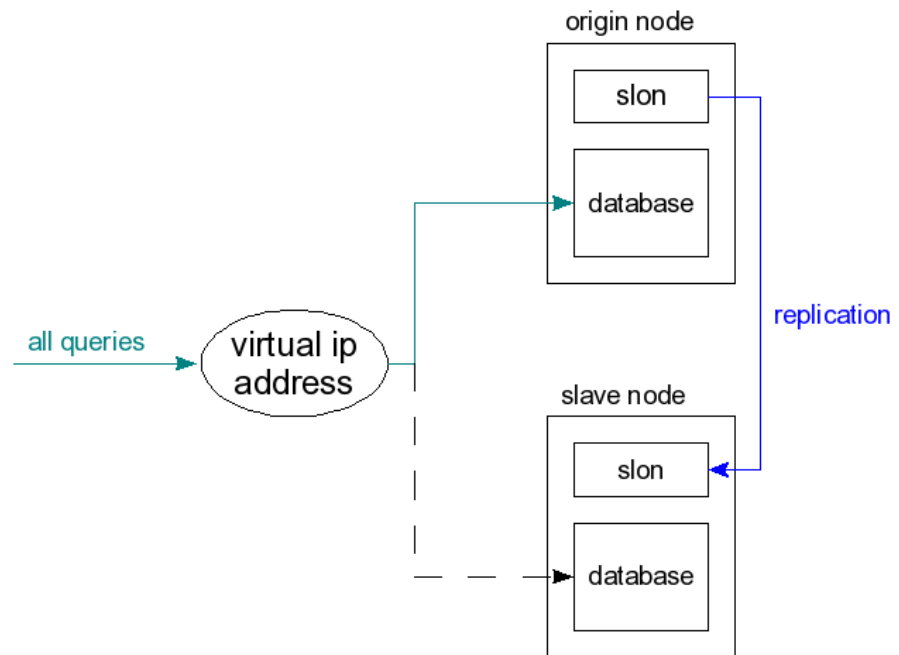
## **5 SLONY-I**

The main object for the LT2 project when it came to database safety was to have a working replication cluster to store all gathered weather observations. As described in previous sections, Slony-I was chosen as the preferred replication implementation, and in this section the process of building a Slony-I based replication cluster from scratch is described in detail. All installation and configuration was done in a weeks time at the LHMS premises in Vilnius, Lithuania. The whole installation process included not only the installation and configuration of the replication system, but also

documentation of the installation process, and training the local IT personnel to operate the replication cluster. The documentation also included Slony-I manual written especially for the LHMS, containing all the necessary information on how to work with the replication system.

## **5.1 Installation and Configuration**

As mentioned above, the overall objective was to have a function database replication for one database containing weather observations. The practical means how this objective was satisfied were to install a Slony-I based asynchronous replication engine to the database system in question. The system consisted of a total of two nodes connected with 100BaseT-ethernet cabling. The installation platform was the latest snapshot of Debian GNU/Linux 4.0 (Etch) with kernel version 2.6.17, PostgreSQL version 8.1.5 and Slony-I version 1.2.1. A hardware RAID controller was used to create a RAID 1 (mirroring) system with two hard disk drives, both 146GB in size, to both nodes. A virtual IP address was created to point one of the serving nodes, the origin node. The purpose of this is that if the master node fails, the virtual IP address is moved to point to the previously subscribing node, and all the clients using the database can still connect to that same IP address although the database is actually serving in another node. The replication scheme is illustrated in Figure 7.



*Figure 7. The actual replication scheme*

In Figure 7 all queries, read or write, are directed to a virtual IP address. The virtual IP will then redirect the queries to the node currently serving as the origin node. The origin node will then replicate the data to the subscribing node. In case of a machine malfunction, the replication cluster would perform a failover, which means that the node previously acting as slave would be promoted to origin. After failover the virtual address would be moved to point to the new origin node, and the clients using the database would continue using the database, with only a minor interruption to the service caused by the switching of roles. Since the amount of data that will be read from the database is rather small, no load-balancing system was implemented at this point.

#### 5.1.1 Prerequisites

The actual installation of PostgreSQL and Slony-I binaries was done by the other FMI IT professionals collaborating in the installation process. PostgreSQL was installed from packages postgresql-8.1, postgresql-client-8.1 and postgresql-contrib-8.1, and the Slony-I files were from packages slony1-bin, slony1-doc and postgresql-8.1-slony1.

After the Slony-I binaries were installed, a replication user was created. Only this user has rights to modify the replication system. To make the management of the cluster easier, a system user and a database user were created, both called `slony1`, to both nodes participating to the replication. The database user was created as a superuser, since the replication system must have access to all tables and sequences in the database. The next thing to do was to grant database access to the newly created user to both nodes in the cluster. This was done by modifying a file called *pg\_hba.conf* at both nodes and adding the proper hostmasks and setting the authentication method as md5, which means that the password will be encrypted with md5 algorithm. Finally three more files were modified or created. The first file was called *.pgpass*, and it was created to the home directory of the `slony1` user. This file contains password information for the `slony1` user, so that they do not need to be separately specified when accessing the database. This also enforces the security of the system, since passwords do not need to be specified as command line options. The files permissions were changed to read-write (0600). The second file was called *pg\_services.conf*, and it was created to the PostgreSQL server configuration home directory, */etc/postgresql/8.1/main*. It contains information on hosts and databases, and it will be referred to when connecting to different databases at different hosts. The third file was called *sl\_params.pm*, and it contains a few configuration options such as the names of the nodes participating in the replication and the name of the replication user. This file is used by all the Slony-I configuration scripts used later on so it must be configured accordingly before running the scripts.

After the user information was correctly set up, a language called PL/pgSQL was created to all replicated databases. This was done inside the database with SQL command `CREATE LANGUAGE`. The language was also created to database `template1`, so later on all new databases will automatically contain the language.

### 5.1.2 Creating a Cluster

When the user information was correctly configured, and the slony1 user could successfully connect to all nodes and all databases, the actual replication cluster was created and configured.

The first step that was done was the actual initialization of the cluster, which created the schema that contains all the Slony-I specific relations. During the initialization process the origin node and the subscribing node were also defined. The initialization required four slonik commands to execute successfully: *init cluster*, *store node*, *store path* and *store listen*. A perl script called *slony1\_init\_cluster* which executed all these four commands was used. Before executing the commands mentioned above, it run a bundle of commands called a preamble, which must be executed every time configurations are made through slonik. The preamble simply consists of the clustername, and the node information (hostname, database, username, password). The preamble tells slonik how it can reach all nodes of the cluster. After the preamble and the set of four commands were successfully executed, tables *sl\_listen*, *sl\_node* and *sl\_path* were populated with the correct information.

When the cluster was initialized, the slon processes for each node and database were started. The default configuration file for slon was used, except that debug level was set to zero and all messages were directed to syslog. Slon also required the clustername, database name, hostname and username as command line parameters, but as all this information has previously been added to *.pgpass*, *sl\_params.pm* and *pg\_services.conf*, they were referred from there and therefore were not visible at the process listing. A perl script called *slony1\_start\_slon* was used to aid the starting of the slon processes. After the slons were successfully started, the initialization of the cluster was complete. The slons could communicate with each other, and the logs showed clear signs of successful syncing between the processes. No replication was being done at this moment though, as no relations had been added to the replication. The process of adding tables to the replication cluster is described in section 5.1.3.

Since slonik command syntax is not very user friendly, a number of perl-scripts have been written by me. These perl scripts are basically slonik commands wrapped in perl, and they make the configuration and modification of the replication cluster much more flexible since they allow command line arguments to be used and therefore one script can serve multiple different clusters. The opposite of this would be to write one script for one operation for one cluster, which can be an exhausting task if many clusters exists.

### 5.1.3 Adding Relations to Replication

After the cluster was successfully initialized, relations were added to replication. Before a table or a sequence can be added, a replication set must be created. This was accomplished with slonik command *create set*. After the set was created, tables and sequences were added to that set with slonik commands *add table* and *add sequence*. Once all tables were added, the set could be subscribed to replication with command *subscribe set*. After the subscription, no more relations could be added to that set. Instead, a new set must be created and then merged with the original set.

Three scripts have been written to aid the task of adding relation to replication, and they are called *slony1\_add\_tables*, *slony1\_add\_sequences* and *slony1\_scan\_for\_tables*. The first two scripts need the clustername, schema name and table name as command-line parameters. They are used to manually add tables or sequences to replication. The third script can be used to automatically add tables to replication sets, and it requires no command line options. All three scripts limit the size of a single replication set to a maximum of ten relations. If a set is full, i.e. it contains ten relations, the scripts create a new set and start adding relations to that new set. If a set is not full, the scripts will create a temporary replication set, add tables to that set and after the set has been successfully subscribed to both nodes, the scripts will merge that temporary set with an existing replication set.

The functionality of the third script, *slony1\_scan\_for\_tables*, is designed so that it needs no user intervention when running. The script automatically scans for tables and sequences that are not replicated and tries to add them to a replication set. It also scans for tables that have been physically



removed, but not removed from a replication set. If such a table or sequence is found, the script removes the information of the table or sequence from the Slony-I catalogues. During the configuration process at LHMS the script *slony1\_scan\_for\_tables* was configured to run from cron every night at 2:00 am, to ensure that all newly created tables and sequences are added to replication, even if the creator of the table or sequence forgets to do it. The moment of the execution of the script was carefully considered, and it was chosen to run at night because at night the script does not interfere with neither other maintenance tasks nor the work of the IT people taking place during the day. Once all required tables and sequences were added to replication sets and the sets were subscribed, the final thing to do was to confirm the replication flow between the nodes. The testing of the functionality is described in the next section.

## 5.2 Testing Functionality

Once Slony-I was successfully configured and the tables were added, the functionality was tested. A good habit is to verify that the configuration is identical in both nodes, which can be done by querying the tables *sl\_table*, *sl\_sequence* and *sl\_set* by hand or by executing the scripts *show\_table*, *show\_sequence* and *show\_set* or by running the script *check\_consistency*. The last script compares the row of the tables mentioned above between nodes.

After the configuration process was done, the script *check\_consistency* confirmed that the cluster was correctly configured at both nodes, so a real life test was performed. A single row of data was inserted at origin node and it was successfully replicated to subscribing node in a few seconds. The configuration of the replication system was at last successfully completed! The final thing to do was to ensure that the slon processes are started automatically after rebooting of a server. A script called *slony1-replication* was copied to */etc/init.d* and symbolic links were created to runlevels 1,2,3,4 and 5. When this was done the subscribing server was rebooted and after it came back online the functionality of the replication was confirmed.

At this point the replication cluster was fully functioning, and the initial data copying was done, so the contents of the databases were identical. That does not mean that the work is done though; a replication cluster requires regular monitoring, since the replication is vital part of the security of the database system. The means of monitoring and maintaining the cluster are described in the following section.

### **5.3 Monitoring and Maintenance**

After the initial configuration, the replication cluster must not be left to its own. It requires frequent observation and maintenance, because basically it is the last line of defence against machine failures and such, and in a dire situation the replication must be working. This section deals with the different methods of controlling, monitoring and configuring the existing Slony-I replication cluster. Section 5.3.1 describes the necessary procedures that must be done when a subscribing node has to be promoted to an origin node.

#### ***5.3.1 Switchover and Failover***

Perhaps the most important property of an asynchronous replication system is the failover property. Failover means that when a node is unable to serve users, due to hardware failure or such, another node belonging to the replication cluster can take over the duties of the failed node. Generally failover can occur automatically, or it may need manual intervention by the administrators. In Slony-I's case it must be done manually, since the creators of the software believe that a decision of moving the origin to another node is too important to be made by an application.

Slony-I has two options for failover mode; one option is to do a switchover, which simply moves the origin node to another node previously acting as a subscriber. A successful switchover requires that all nodes are online and responding, and a typical situation where a switchover is used is when the origin node is scheduled for downtime (maintenance). The second option for switching roles between the nodes is the actual failover command, which also moves to origin to a new node, but it also removes the old origin completely from the replication cluster. The failover command should be

used only when the origin node is not responding, because once the command is issued the broken node is removed from the cluster and it has to be reconfigured manually if it is to serve as an origin node again.

A Slony-I switchover can be performed for a single replication set. In most cases however the most secure and consistent way is to do a switchover for all sets and all clusters at the same time. The switchover-command actually consists of four slonik commands, (1) the set that is to be transferred is locked with *lock set*, (2) the transferring of the set is confirmed with *wait for event*, (3) the origin status of the set is moved to the other node and implicitly unlocked with *move set* and finally (4) the moving of the set is confirmed with *wait for event*. The script *slony1\_switchover* executes all these commands. The script also transfers the origin of all sets of all clusters at the same time.

The failover, as mentioned above, is a much more coarse way of transferring the origin status. It consists of two slonik commands, (1) the actual moving of the origin status to another node is done with command *failover*, and (2) the failed node is dropped from the replication cluster with command *drop node*. The script *slony1\_failover* executes the required slonik commands. When recovering from a failover, several commands must be entered. First, the node must be initialized and listening paths must be created with slonik commands *store node*, *store path* and *store listen*. These commands create the Slony-I schema and add the necessary configuration the nodes need to contact each other. Next, the sets that are subscribed at the current origin (old subscriber) must be also subscribed to the current subscriber (old origin). This is done with slonik command *subscribe set*. The perl scripts *slony1\_recover\_from\_failover* and *slony1\_subscribe\_set* can be used to perform the tasks described above. Sometimes it is necessary to alter a database schema in an existing replication cluster, and that alteration process is described in the next section.

### 5.3.2 Modifying Configuration

Although a well designed database schema does not usually change very often, inevitably there comes a time when new relations have to be added or old ones modified. In order for Slony-I to recognize the schema changes, all DDL commands affecting the already subscribed relations must be run through *slonik*, which then executes the changes at the database. Modifying the database schema directly can in the worst possible case stop the replication of the relation.

A benefit of executing the changes through Slony-I is that the changes executed at the origin node are automatically propagated to all subscribing nodes. This makes managing the databases easier. The downside (there is one of course) is that when the *slonik* command *execute script* is executed, the command takes exclusive locks on all replicated tables. While these locks are granted, no other than a read query can access the tables. A perl script called *slony1\_execute\_script* can be used to execute DDL commands.

Monitoring is one of the key factors that constitute a working replication system. Slony-I has no built in monitoring solutions, in fact it has only a single view that presents the state of the cluster. Fortunately the state can be queried by other means, and the monitoring is described in the next section.

### 5.3.3 Slony-I Monitoring

A number of scripts has been written to aid the monitoring of the Slony-I configuration. Scripts *show\_tables*, *show\_sets* and *show\_tables* all scan the corresponding system table on the node they are executed at and print out the results. The script *show\_nodes* queries the tables *sl\_set*, *sl\_node* and *sl\_subscribe* and prints out the current origin node of the replication. Script *show\_lag* queries a view called *sl\_status*, which is practically the only source of valid information concerning the replication delay between the nodes, and retrieves the value of the column called *st\_lag\_num\_events* and prints it. This value, which is only applicable at the origin node, is the amount of sync events the subscriber is lagging.

The script *check\_consistency* takes a row count from tables *sl\_table*, *sl\_sequence*, *sl\_set* and *sl\_subscribe* at both nodes and compares the results. This is a very unsophisticated check, but in most cases it is the simplest and most efficient way to find the problem, since if the configuration is faulty in most cases the reason is that some relations are configured at the origin but for some reason they have not propagated to the subscriber. This kind of misconfiguration situation is easily detected with the row count method.

Efficient replication is often in theory and in practice a hard thing to accomplish, due to its complex nature. Slony-I installation is very precise and often very awkward, but once the installation is done an administrator rarely needs to intervene with the replication especially if the database schema is stable, i.e. DDL changes are made rarely. Slony-I has proven to be very robust and scalable replication software, even with hundreds of tables and sequences. The next section describes how PgPool-II, a PostgreSQL based load balancing solution, is installed, configured and performance tested.

## 6 PGPOOL-II

If a multi-node database system has a functioning replication, meaning that the system has at least two databases with equal contents, introducing a load balancing solution to that database system is often moderately easy and requires no additional changes to the database architecture itself. In section 6.1, PgPool-II is installed and configured, and in section 6.2, the load balancing capabilities of PgPool-II are tested by making a series of performance tests on a simulated environment. Although the LT2 projects' needs are taken into consideration when choosing the load balancer, the actual implementation will not be installed to the LT2 database system, but the results gathered will be used as guidelines in the future when the need for load balancing might emerge.

## 6.1 Installing and Configuring

Since PgPool-II was not available via Debian's package manager, it had to be installed from a source packet. PgPool-II requires gcc version 2.9 or higher, GNU make and since it links with the libpq library, the library must also exist on the machine used to build the program. After all these tools necessary for the PgPool-II installation were installed, the PgPool-II source file were downloaded from the PgFoundry webpages. The default installation directory of pgpool was */usr/local*. The installation created two example configuration files to */usr/local/etc*, one for the pgpool itself and one for pgpoolAdmin, a graphical user interface for the load balancer. The pgpool configuration file (accordingly named as *pgpool.conf*) was modified so that PgPool-II will listen to all active network interfaces. The load balancing mode was initially set to master/slave, since the two first performance tests used that mode. The other configuration file called *pcp.conf* contains an md5-encoded password of a user that is allowed to access the pgpoolAdmin program.

When the installation was complete, it was time to run some tests on the system. The testsuit consisted of a total of four different stress tests, or benchmarks, and they are described in the next section.

## 6.2 Performance Tests

The primary function of a load balancer is to distribute load efficiently between a number of servers. In order to find out just how efficient PgPool-II is, the system was put to the test. The benchmarks were run on a system consisting of four machines serving PostgreSQL databases. All four servers were nearly equal in performance, with an average processor speed of 2 GHz, IDE hard drives and 1Gb of RAM. Since the purpose of the benchmarks was not to measure the absolute speed of the load balancer, but the relative increase or decrease in performance caused by the load balancing solution, the hardware was sufficient enough for the tests. The contents of the databases were initially replicated with Slony-I, and the first two performance tests were run with PgPool-II's master/slave mode. The two latter tests were run with PgPool-II's parallel query mode, which requires

horizontal fragmentation of the data, so for the parallel query benchmarks the Slony-I replication was removed and the data was distributed accordingly. The following section describes the theory behind the performance tests in detail.

### *6.2.1 Theory Behind Tests*

As mentioned previously, the whole benchmarking process consists of a total of four different stress tests. All tests are run on one to four servers, and they measure only one thing, the maximum number of transactions per second (tps) the whole multi-node database system can perform. The tests can be divided into two categories: tests run using PgPool-II's master/slave mode and tests run using the parallel execution mode with horizontal fragmentation. With both load balancing modes the performance (in transactions per second) of two different query types were measured: the speed of an index scan and the speed of a sequential scan. These scanning methods, index scanning and sequential scanning, are the two fundamental methods for a database to retrieve information from a disk or a memory segment, and therefore they are both tested in order to find out what type of queries benefit the most from load balancing.

Sequential scan is the most basic data retrieval operation, and it simply means that when some rows of a relation are queried by a user, the database will read the whole relation from disk in order to find the rows the user wants. The important thing to understand is that the sequential scan will read the entire relation, even though it might find the information user requires from the beginning of the relation. An index scan uses an index to retrieve information. An index is just a file that is structured so that the database can using the file quickly find one piece of information from the actual relation, in fact the index only points out where in the disk a specific piece of information physically exists. Both file retrieval types have their advantages and disadvantages, but the basic usage pattern is such that depending on the size of the relation, if more than 10% - 15% of the relations rows are to be fetched it is often more efficient to use a sequential scan, due to the way the rows are physically ordered on the disk. Of course if a relation does not have an index, a sequential scan is the only way to retrieve

information. If less than 10% - 15% of a relations rows are to be fetched, an index scan is usually used. An index scan is normally several orders of magnitude faster than a sequential scan, a good example is a dictionary: finding one word in a dictionary is much faster using an index than shuffling through the the whole book. The performance tests done in section 6.2.2 and 6.2.3 test the speed of an index scan and the speed of a sequential scan with different load balancing methods, and different amount of concurrent users.

The performance tests are run in a rigorous fashion: for each of the four tests (master/slave mode with index scan and sequential scan, parallel execution mode with index scan and sequential scan), the testing procedures are the same. The idea is that first the load balancing system is tested with just one node and no load balancing in order to have a base level of performance where to compare the results drawn from the other tests. After the first run, load balancing is added to the system, but just for one machine so that the gain in performance caused by the connection pooling can be measured. After the second run with the connection pooling, a second node is added to the system and the performance of the system is tested again. After that run a third node is added and tested and finally a fourth node is added and the maximum performance of the four-node load balancing system is tested. All these tests are also performed with different number of concurrent users: 1,2,5,10,15,25,40,60,90 and 120 users. For each number of concurrent users the stress test is run for three times, and the average of the three runs is the final result for that particular run.

To summarize the theory, first the performance of the first subcategory, master/slave mode with index scan, is tested with no load balancing and only one concurrent user. This is repeated three times in order to have a reliable result. The result is a given amount of transactions per second, and the result is saved. After running the test with only one user, the same tests are run again with two concurrent users, and the average of three runs is again calculated and saved. After the index scan test with only one node has been performed with 120 concurrent users accessing the database, a load balancing solution using only connection pooling is added to the system and the same tests are run again. Thereafter the second, third and fourth nodes

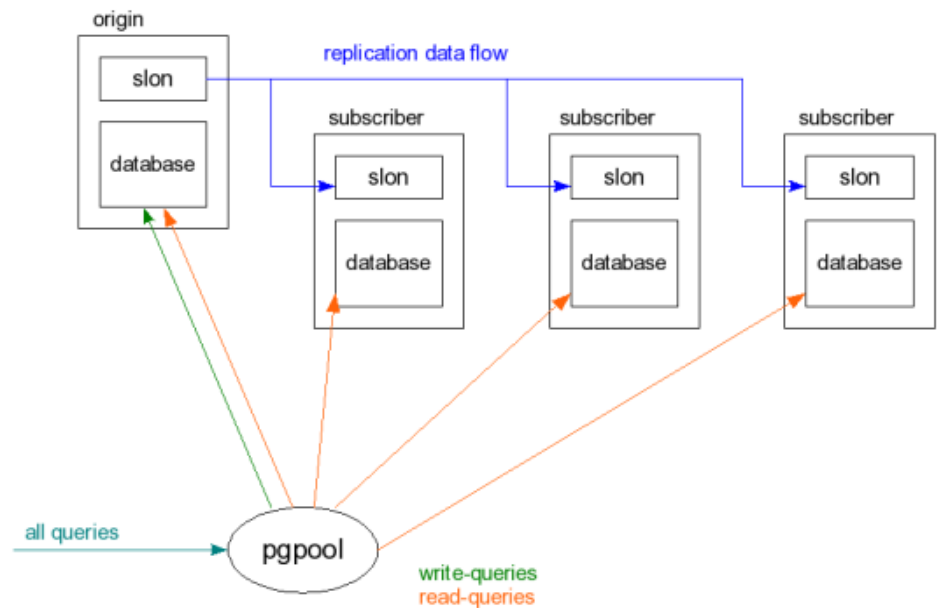


are added, and the same tests are run for each number of nodes. After the final test run for the master/slave index scan measurement has been finished, all procedures are repeated for sequential scans tests and for parallel execution mode tests. All tests are run in read-only mode, since Slony-I does not allow writing of data to slave nodes, and in order to have consistent results between the different PgPool-II load balancing modes the parallel execution test uses the same queries as the master/slave test.

From the results of the tests a few observations can be determined instantaneously: the optimal amount of servers for a certain number of concurrent users, data retrieval method, index scan or sequential scan, more suitable for load balancing and load balancing mode, master/slave or parallel execution mode, more suitable for production environments. In section 6.2.2 the practical issues with master/slave benchmark are described, the actual tests are performed and their results are presented.

### *6.2.2 Master/slave Mode*

In master/slave performance test queries are distributed as whole to different servers in the load balancing server pool. A program called pgbench was used to initialize the benchmarking database and also to perform the actual tests. When initialized pgbench created four tables to each node and populated them with one million rows. Pgbench was also used to simulate a different number of concurrent users, and the amount of queries one simulated user executes can be specified separately. Using pgbench in index scanning tests the value of a single column was read by one user one thousand times, and in sequential scanning tests five times. The overhead coming from the connection establishing was added to the result. As stated earlier, the first performance test measured the speed of an index scan, and the second test measured the speed of a sequential scan. Figure 8 presents the benchmarking system for the master/slave mode.



*Figure 8. Benchmarking system for master/slave mode.*

Figure 8 illustrates the master/slave benchmarking architecture used in the first two tests. PgPool-II frontend received all queries, and redirected write-queries to the origin node. As the test are run with read-only mode, the green arrow does not actually exist and all queries are distributed to all servers. When the database used for the benchmarking was created and populated, and the origin node was initialized through slonik, Slony-I replicated the contents of the database to all three slave nodes. After initialization the replication data flow drained since no changes were made to the database. Figure 8 does not illustrate the very first test run which was performed with a single database and no load balancing. The results of the first benchmark are presented in Figure 9.

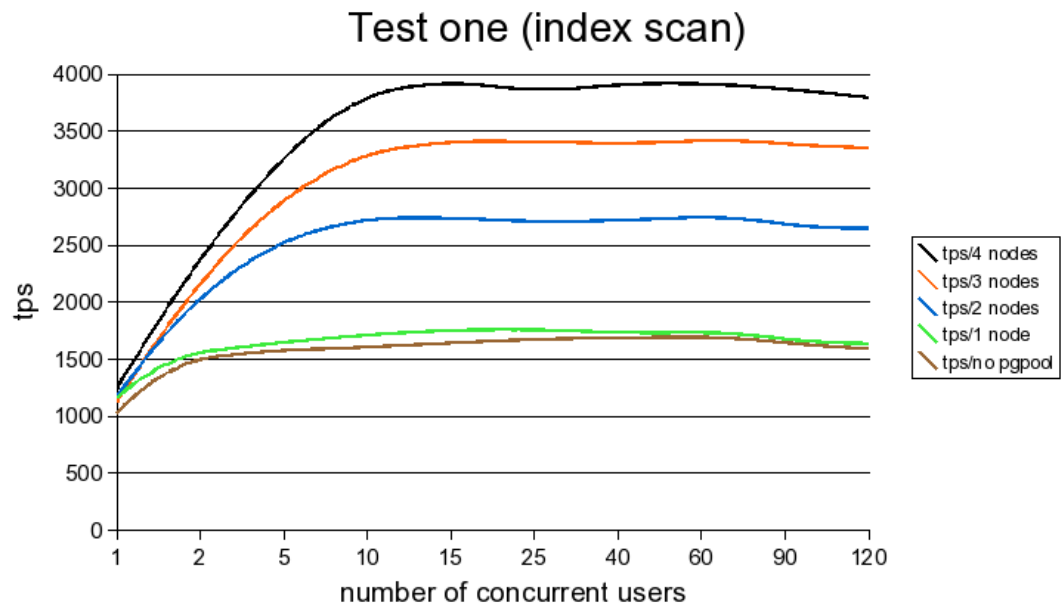


Figure 9. Results for the first benchmark (index scan).

Figure 9 shows the speed of an index scan. The Y axis contains the quantity measured, transactions per second (tps), and the X axis contains the number of concurrent users. The lowest line, marked with brown colour, presents the speed of the system without any load balancing. As can be seen, the amount of tps stays nearly constant all the time despite the growing number of users. The green line presents the speed of the system when PgPool-II is used as a connection pooler. The connection pooling improved the performance a minuscule amount, the biggest difference was achieved when the number of concurrent users was between 10 and 40. The blue line presents the first test with the actual load balancing using two nodes. As can be seen, the performance was greatly improved when compared to the previous run with only one node. The performance increased steeply until the number of users was around ten, and thereafter the performance was fairly steady. The orange line presenting the performance of a three node system behaves similarly, the performance increased sharply until about ten concurrent user, and stayed quite stable after that. As can be expected, the maximum performance was achieved when all four nodes participated to the load balancing. This four-node test is marked with a black line, and as in previous test runs the increase in performance stopped when the number of concurrent users was ten. The results are furthermore analysed in section 6.2.4.

The second performance test measured the speed of a sequential scan. The benchmarking schema was the same as in the first test with the exception that indices were dropped from the queried tables so that the database was forced to use sequential scans. The results for the sequential scan benchmark for master/slave mode are presented in Figure 10.

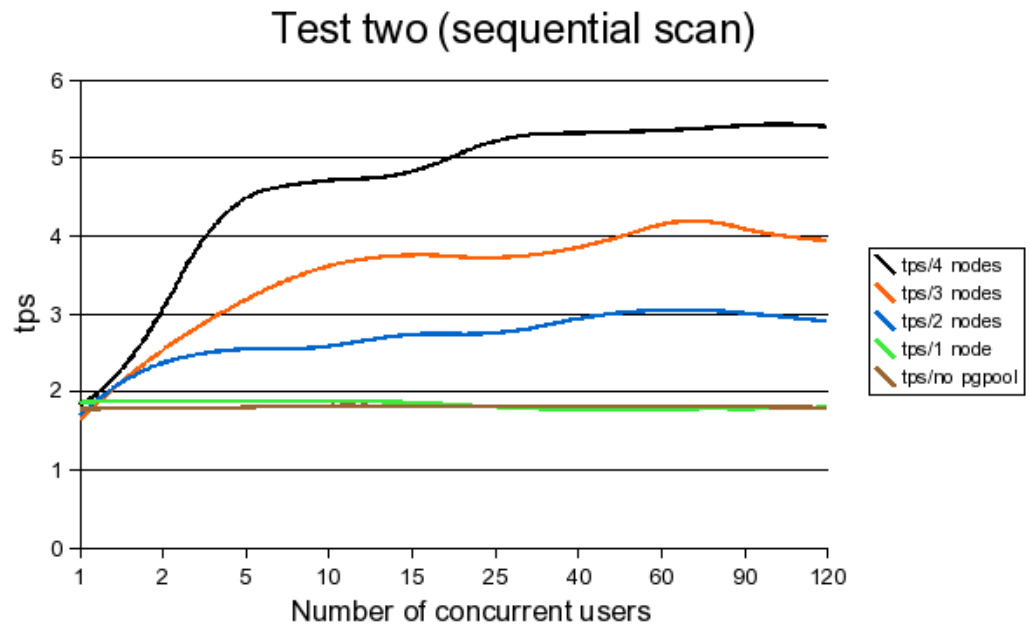


Figure 10. Results for the second benchmark (sequential scan)

Figure 10 shows the results for the second benchmark. The Y axis contains the result, transactions per second, and the X axis contains the amount of concurrent users. The colouring scheme is equal to Figure 9, again a brown line presents the performance of the system without any load balancing, and a green line is used to present the performance of a system using connection pooling. As can be seen, the brown and green line are nearly identical, so no performance gain was received when using connection pooling with sequential scans. A blue, orange and black line presents the performance of the system with two, three and four nodes, respectively. A clear performance boost can be seen, as the the maximum performance of the load balancing system with four nodes is nearly three times faster than the speed of a single database with no load balancing. Again the results are analysed and interpreted in section 6.2.4. In the next section the parallel execution mode of PgPool-II is explained and tested.

### 6.2.3 Parallel Execution Mode

PgPool-II has a functional mode called parallel execution mode, which used to distribute horizontally fragmented data to different servers. This is a fundamentally different mode when compared to the master/slave mode with Slony-I as the replication solution: with master/slave each database contain a whole copy of the data, but with parallel execution mode each database contains only a fragment of the whole data, and only the cluster combined contains the whole data. The horizontal fragmentation has a great benefit over the master/slave replication; since the whole dataset is divided between all the servers in the pool, the more servers the pool contains the less data one individual servers has to host. Especially sequential scans should benefit from a smaller set of data.

The performance testing of the parallel execution mode consisted of an index scan test and a sequential scan test. The testing procedures were similar to the master/slave tests with a few differences: the tests were only run with two, three and four nodes, since parallel execution required at least two servers. The data was also redistributed every time a new server was added to the server pool. Initially with two servers the data was distributed so that the first server contained half of the relation of one million rows, and the second server contained the other half. When a third server was introduced to this system, the data was distributed so that each server had one third of the data, and with a four node system each node had one fourth of the data. The data initialization and the actual testing was done with the pgbench program also used in the previous tests. Since the data pgbench inserted when initialized was only integer numbers from one to one million, the data distribution rules were easy to configure. In a real life situation, when the data is more dynamic, the distribution rules can be very hard to come up with. Figure 11 illustrates the parallel execution mode benchmarking system with all the four nodes.

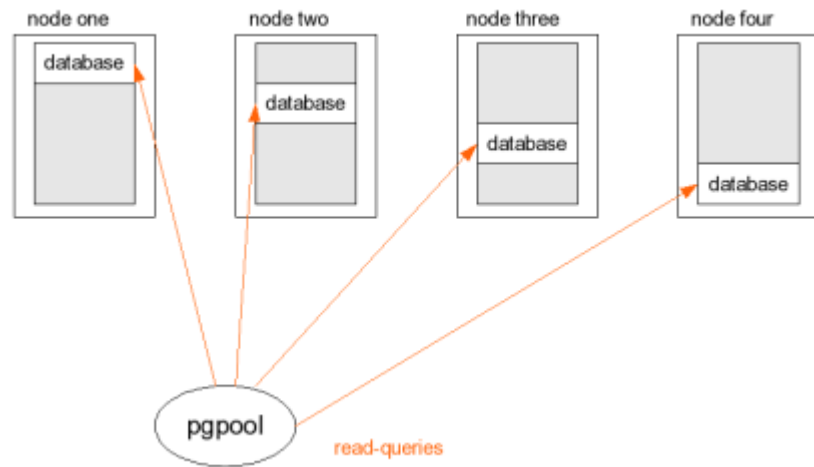


Figure 11. Benchmarking system for parallel execution mode.

Figure 11 illustrates the horizontal fragmentation used in the parallel execution mode. Node one contains the topmost quarter of the whole data set, so all queries accessing that data are directed to node one. Node two has the second quarter, node three the third quarter and node four has the fourth quarter of the total data. When combined these four nodes make up the whole database. The third performance test (first with parallel execution mode) tested the speed of an index scan. The testing procedures were the same as in test one. The results of benchmark three are presented in Figure 12.

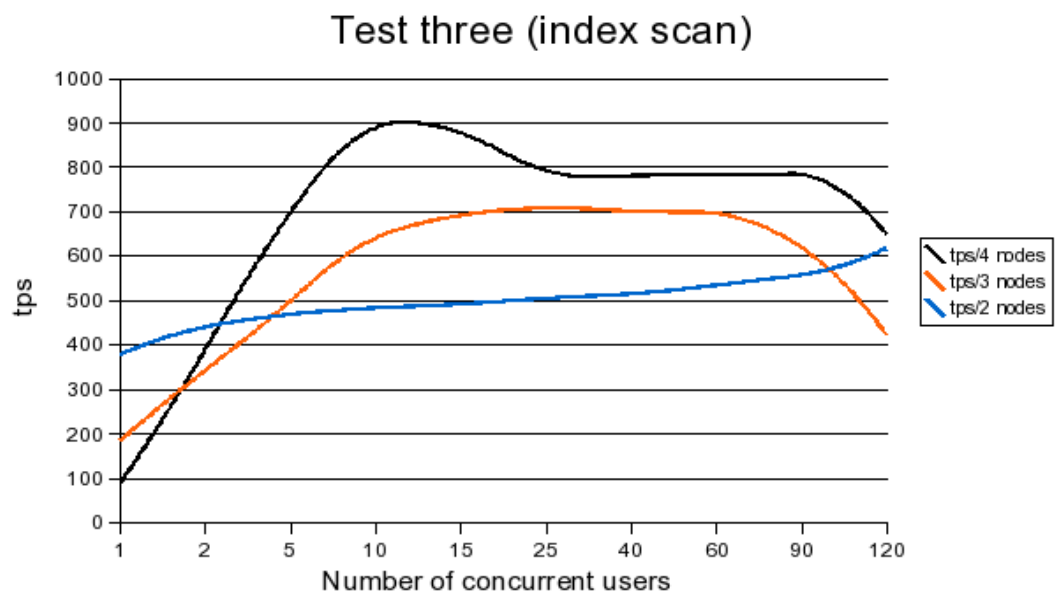
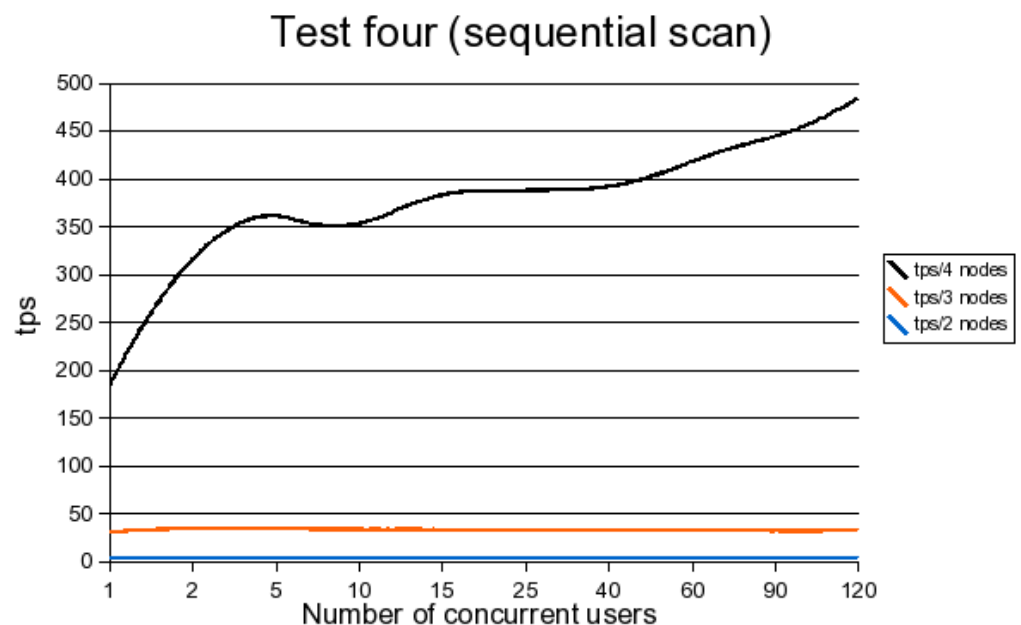


Figure 12. Results for the third benchmark (index scan).

Figure 12 illustrates the results for the third benchmark. The colouring scheme is again same as in the previous graphs: a blue line represents performance of the load balancing system with two nodes, an orange line means three nodes and a black line represents four nodes. The system with two nodes had a quite steady performance, the only surprising thing was that the system was able to perform only one tenth of the amount of transactions per second than the master/slave mode index scan. The performance with three nodes introduced a strange phenomenon: the tps value with one to five concurrent users was much lower than with the system consisting of two nodes. After five users the three-node system outperformed the two-node system until about 90 concurrent users, where the performance dropped dramatically. The same behaviour was seen with the four-node system, the performance with just a few users was very bad, but quickly the tps value rose, and then again with more than 90 concurrent users the performance dropped. These results clearly differ from the first benchmark, which tested the speed of an index scan with master/slave mode. The results are furthermore analysed in section 6.2.4.

The fourth performance test measured the speed of a sequence scan with PgPool-II's parallel execution mode. The testing procedures were the same as in the previous benchmarks. The results for the fourth and final benchmark are presented in Figure 13.



*Figure 13. Results for the fourth benchmark (sequential scan).*

In Figure 13 the results for the fourth benchmark are presented. Again the quantity measured was transaction per second (tps). A blue line presents the performance of the system with two nodes, an orange line presents a three-node system and a black line presents a four-node system. The performance of the two-node system was quite modest and very similar to the master/slave sequential scan test, but when a third node was introduced the performance increased radically. The performance of the two- and three-node systems was quite stable, and no significant increase or decrease was found when the amount of concurrent users increased. The performance of the four-node system was quite different from the previous two. The tps value was considerably larger than in other tests and the performance also increased throughout the test. The results are analysed in section 6.2.4.

With databases parallel load balancing is often considered as more efficient than distributed load balancing, but the parallelism comes with a price: in order to efficiently split queries and distribute them to different servers the load balancing solution has to examine and possibly mangle each incoming query. When queries come in by the thousand, the process of examining every incoming query can become a bottleneck for the whole system. PgPool-II's horizontal fragmentation introduces another complicating factor: since each server has only a fragment of the data, when a single node malfunctions the cluster can not function fully extent because the part of the data the failed server hosted is missing. A backup from the whole data is also hard to make, since a backup process has to be run at all serving database servers.

The whole process of benchmarking was a tedious task, as the total number of individual tests was rather large. The results from the benchmarks were quite interesting, as they varied very much between the different PgPool-II load balancing modes. The results are analysed and interpreted in the next section.



### 6.2.4 Analysis

The results from the four performance tests were quite intriguing: on one hand, the results from master/slave mode were quite as expected, but on the other hand the parallel execution mode produced very unexpected results. In this section the results are first individually analysed, and then the results from the matching techniques, index scanning and sequential scanning, are compared. A summary of the results gathered from all four tests are presented in Table 2.

Table 2. A summary of the performance test results.

benchmark	tps/4 nodes	tps/3 nodes	tps/2 nodes	tps/1 node	tps/no pgpool
<b>one</b>	3325.98 (13.6%)	2926.25 (20.5%)	2428.33 (49.6%)	1623.16 (4.6%)	1551.98
<b>two</b>	4.47 (31.5%)	3.40 (28.8%)	2.64 (45.1%)	1.82 (1.1%)	1.80
<b>three</b>	652.78 (22.2%)	534.07 (7.2%)	498.05		
<b>four</b>	371.00 (1050%)	32.19 (960%)	3.04		

In Table 2, the four different tests are located at the leftmost column, and the topmost row contains the results for a given number of database servers in the load balancing server pool. The results are given as average transactions per second, and the percentage increase in performance compared to the previous level has also been calculated. Tests three and four were not run with only one node or without PgPool-II.

As mentioned, the results from the master/slave mode benchmarks were not very surprising. In the first benchmark the speed of an index scan was measured, and the load balancer was clearly improving the overall performance of the system. The highest tps value was achieved with a four-node system, the performance of that system compared to the system with no load balancing was 114% higher. The performance of the four-node system was 13% higher than the performance of the three-node system. The largest individual gain in performance happened when the load balancing system with just connection pooling was added a second node, the performance was increased by 50%. When connection pooling was added to the system with no load balancing, the systems performance increased only

4.6%. As can be seen from Figure 10, the performance of the system with no load balancing and that of the system with just connection pooling reached their maximum performance quite rapidly, after only two concurrent users. In other words, both systems were saturated after two users and were not able to perform any better with more concurrent users. This is a clear disadvantage since if the data and business logic is archived in the database, it is more than likely that more than two concurrent users are accessing the database. With the two-, three-, and four-node systems the saturation point was much higher, around ten to fifteen users. It is safe to say that in the case of index scans the maximum performance is achieved when the number of concurrent users is more than two, if no load balancing is used, and more than 15, if load balancing is used. One added server increased the performance an average of 28%, although the percentage increase diminishes with each new added server.

The second benchmark measured the speed of a sequential scan with PgPool-II's master/slave mode. This test clearly showed the difference in speed between an index scan and a sequential scan: an index scan was nearly one thousand times faster. Again, the maximum performance was achieved with a four-node system, the performance gain compared to a system with no load balancing was 148%. The single biggest difference between test runs was achieved when a second node was added to a system with only connection pooling, there the performance was increased 45%. Surprisingly the connection pooling method had little or none effect on the performance of the system. As explained earlier, connection pooling means that when started, an application immediately spawns a number of processes to server users, even though there may not be enough users to access all newly started processes. This pre-spawning saves resources, since when a users connects to a database, the database management system does not have to spawn a new process and spend resources doing that. The reason why the second benchmark did not benefit from connection pooling is that because sequential scanning is so slow and resource-consuming, the overhead of spawning a new process is nothing compared to the overhead caused by the sequential scan. The second performance test did not produce clear saturation points like the first one did. With no load balancing and only connection pooling, the system did not perform any

better with different amount of users. Figure 11 shows that with a two-node system the saturation point occurred somewhere between 40 and 60 users. With a three-node system the performance was gradually increasing until about 15 concurrent users, and the maximum performance was achieved with 60 to 90 users. With a four-node system the performance increased rather steeply until about 5 users, and the maximum performance was achieved after 25 users. It can be determined that for sequential scans, connection pooling is not a feasible solution, but adding more nodes to the load balancing pool clearly increased performance. One added server increased the performance of the whole system an average of 35%.

The results from the third benchmark were quite surprising. The third benchmark measured the speed of an index scan using PgPool-II's parallel execution mode as a load balancing solution. The parallel execution mode means that the data has been partitioned to different servers, and a single server has only a fraction of the total data. What one would expect from this test is that the parallel execution mode would outperform the master/slave mode, since each server hosts only a fraction of the data the data retrieving should be faster. The results however strongly suggest that index scans using parallel execution mode are much slower than index scans using master/slave mode. An average tps result for parallel execution mode was only 20% of the result received using the master/slave mode. The performance did not get much better when new nodes were introduced to the system: the performance of a three-node system was only 7.2% better than that of the two-node system, and the four-node system outperformed the three-node system by 22%. The reason for the doleful performance lies within the parallel execution mode itself: every time a query is issued to the PgPool-II frontend supporting parallel execution mode, the query is examined and rewritten, a new database link is created to the node serving the queried data and only after these two procedures are done the actual querying can start. Compared to the minuscule amount of the actual index scan takes, the examination of the query and the opening of a database link take a very long time, so the PgPool-II frontend becomes the bottleneck of the system. As can be seen from Figure 12, when the system was made of three or more nodes, the performance started to drop when a large amount of users were connected to the database, due to the fact that more users

produce more queries and more queries mean more database links. Benchmark three has clearly shown that parallel execution mode does not suit well to situations where fast index queries are a majority of the queries.

The fourth and final performance test produced perhaps the most surprising results. In that test the speed of a sequential scan was tested in a system using PgPool-II's parallel execution mode as a load balancing solution. As in the previous benchmark, the whole data was fragmented to different servers so that each server hosted only a subset of the total data. It would be expected that the system would perform better than the similar benchmark using PgPool-II's master/slave mode (namely benchmark 2), because since each server had only a subset of the data, it should take less time to read the whole data. The results were quite staggering: the performance was quite modest with a two-node load balancing system, but when a third node was added the performance increased by 960%. The performance gain was even larger when a fourth node was added, 1050%. The explanation of this dramatic increase in performance is simple: with a two-node system each node hosted 500 000 rows, which was too much for the consumer level hard disk drive subsystems, which practically choked when they had to read all the rows. With a three-node system each node hosted only around 330 000 rows, which apparently crossed some threshold in the disk subsystems ability to read data and the disk was able to read the rows in a much faster fashion. Adding of the fourth node reduced the per-node amount of rows to 250 000, and again the rows could be read even faster. Clearly if even more nodes were added the performance would have continued to increase.

Even with these results in hand, it is not easy to decide which load balancing method is better as both methods have their advantages and disadvantages, but at least some conclusions can be drawn: on one hand, if the majority of the queries use index scans to access the database, the obvious choice would be to use the master/slave mode of PgPool-II. The performance of the four-node system using master/slave mode was 410% better than the performance of a similar system using parallel execution mode. In fact, the master/slave mode outperformed the parallel execution mode in every section when it came to index scanning. On the other hand, if the majority of the queries use sequential scans, parallel execution mode is clearly much faster than the master/slave mode. The performance of the parallel

execution mode with a four-node system was 8200% better than the performance of the same system using master/slave mode. In real world the division between the query types is often vague, and therefore also other issues beside the actual performance such as consistency of the data and the facility of backing up the data must be taken into account. On both of these areas the master/slave mode is clearly better, since it offers the option to replicate data with Slony-I, and also backups can be taken from any server in the server pool. The parallel execution mode does not replicate the data, and backups are hard to take since each server has a part of the data. Therefore it can be concluded that overall parallel execution mode, although better in performance in some areas, has more disadvantages than the master/slave mode. The next section closes the whole project; all results from all relevant sections are gathered and summarized and yet again some guidelines are drawn from the results.

## 7 CONCLUSIONS

In this section the results from the previous sections are summarized and guidelines are drawn. First the results gathered from Slony-I installation and configuration are presented and afterwards results from performance tests are presented and evaluated.

Database replication is one of the key things when constructing a high availability database system. In general replication methods are numerous, but for PostgreSQL database management system they are very limited; there are only a handful of different replication implementations available, and few of them are actually stable enough to consider using in a production environment. For the LT2 project's needs an asynchronous replication was chosen, and the actual software was called Slony-I.

The installation of Slony-I was quite simple and no real problems surfaced during the initial configuration. The complexity was introduced when the system was configured for the replication: the theory behind Slony-I is rather complicated and requires careful planning in order to be consistent and fault

tolerant in all situations. The advantages of Slony-I over other competing replication implementations are

1. Once configured, Slony-I is very a robust replication system, and in fact guarantees that as long as the PostgreSQL database itself is running no information is ever lost
2. The basic principle behind Slony-I is simple, and it is by far the easiest replication implementation to install. It is also available in many binary-packaged Linux distributions, and requires no patching of the original PostgreSQL source code.
3. Slony-I is an active product with developers around the world.

The disadvantages are:

1. Although basic in principal, the Slony-I implementation is quite abstract and complex, and the configuration of the actual replication requires careful planning.
2. Slony-I is still a quite new application and therefore has some awkward and illogical features.
3. As open source software, it has no commercial support (at least outside the USA).

As said, the replication systems for PostgreSQL databases are few, and of those Slony-I is the most promising. The advantages listed above outweigh the disadvantages, and overall with careful planning, Slony-I can be (and in fact is used) in critical and high-demanding production environments. As a result of this final project the weather observations gathered at Lithuania are now secured from data loss as the database system at LHMS premises is replicated with Slony-I. The replication system consists of two individual nodes and one replicated database, and in case of an emergency the origin status of a node can be transferred to another node just by running one script. The modification and monitoring of the replication system were also made as easy as possible; overall the heart of the quality control system and the final destination for the weather observations, the database system, is well secured.

Load balancing methods for PostgreSQL are even more limited than replication methods: not counting a few java based solutions, PgPool-II is the only option available. Fortunately PgPool-II is a very versatile load balancer and offers many different load balancing modes.

In this study two different PgPool-II load balancing modes were tested performance wise, and the results were also compared in order to find out which method is superior. One of the modes, master/slave mode was chosen because it would be the best choice if the LT2 project would want to implement a load balancer. The other mode, parallel execution mode, was chosen because supposingly it should give the best overall performance. The results gathered from the performance tests were not uniform: as could be expected, both methods had their own advantages and disadvantages and are aimed to different load balancing scenarios. The results are summarized to Table 3.

Table 3. Load balancing results summarized.

<b>Load balancing mode</b>	<b>Suitable for index scanning?</b>	<b>Suitable for sequential scanning?</b>	<b>Replication of data?</b>	<b>Backup facility?</b>
master/slave	yes, linear performance gain	yes, linear performance gain	yes, with Slony-I	easy
parallel query	no, large performance reduction	yes, exponential performance gain	no	hard

Table 3 contains analysed results from all four performance tests. The leftmost column contains the two different load balancing modes, and the topmost row contains some important features that were measured and perceived during the stress tests. Master/slave mode was a load balancing mode which used Slony-I replication engine as the underlying distributor of data. When this mode was enabled, PgPool-II distributed all write queries to the master server, and all read queries to any of the server in the server pool. Parallel execution mode was a PgPool-II's special load balancing mode, which fragmented the whole data into small subsets to different database servers. Index scanning means that the data in a database is read through an index, while sequential scanning means that the data is simply read from the actual data files on disk. Index scanning is usually many

orders of magnitudes faster than sequential scanning, but both data access methods have their own usage patterns.

As summarized to Table 3, master/slave mode gave a performance boost when data was accessed through an index scan. The performance increased as new nodes were introduced to the load balancing system, and the performance also increased when the number of concurrent users accessing the database increased. The gain in performance was quite linear, each new node in the load balancing system added to the performance roughly the same amount of transactions per second than what was the performance of a single server without any load balancing. The index scanning performance with parallel execution mode was quite wretched, and in fact the parallel execution mode with four nodes had worse performance than the performance of a single, non-load balanced server in master/slave mode tests. The reason is the overhead resulting from the query examination of parallel execution mode.

With sequential scanning the master/slave mode was also improving performance quite linearly, each node introduced to the load balancing system improved the performance the amount it could handle, i.e. when the performance of one server was roughly one transaction per second, the performance of a four-node system was roughly four transactions per second. With parallel execution mode, the performance gain was impressive: the gain in performance when a third and fourth server was added to connection pool was around 1000% for each added node. The reason was that as more nodes were added, each node had to host a smaller fragment of the total data and therefore the reading of the data was much faster.

Another thing to consider with load balancer is that how do they affect the data integrity and protection against machine malfunctions and such. With PgPool-II's master/slave mode the data is safe, since it is replicated with Slony-I. It is also easy to backup the whole database, since each server has a whole copy of the data so the backup can be run on any of the servers. With PgPool-II's parallel execution mode things are not so easy: the data is not replicated between the servers, and each server has only a fragment of the total data. This means that when for example one server is



disconnected, the load balancer can not function because it can not see that specific part of the data. With parallel execution mode the backing up of the whole data set is difficult, since the data is divided between servers. With these factors in mind, it is easy to conclude that PgPool-II with master/slave mode is very robust and reliable load balancer which can be used in situations where the five nines availability is required. On the other hand, parallel execution mode has some advantages but the disadvantages are such that the parallel execution mode is hardly a candidate for situations where high performance and reliability is needed.

## REFERENCES

- [1] LHMS, *Lithuanian Hydrometeorological Institute* (2006), [WWW-document] <http://www.meteo.lt/english/about.php> (Accessed Nov 23, 2006)
- [2] Jari Härkönen, *Cost–Benefit Analysis (CBA) of automating weather observational network in LHMS*, Internal document dated Dec 8, 2006 (Accessed Dec 15, 2006)
- [3] Pauli Rissanen, *LT2 Short Description*, Internal document (Accessed Dec 15, 2006)
- [4] Tom Sanders, *Open source databases '60 per cent cheaper'* (Nov 20, 2006), [WWW-document] <http://www.vnunet.com/vnunet/news/2168971/open-source-databases-slice> (Accessed Nov 22, 2006)
- [5] DB Group, *A Brief History on Databases*, [WWW-document] <http://www.db.web.cern.ch/wwwdb/aboutdb/history/industry.html> (Accessed Sep 28, 2006)
- [6] Quantum RBS, *Top 10 Reasons for Remote Data Backup Solutions* (2005), [WWW-document] <http://www.quantumrbs.ca/top10.html> (Accessed Nov 22, 2006)
- [7] National Research Council (1999), *Funding a Revolution: Government Support for Computing Research*. 1<sup>st</sup> edition. Washington D.C: National Academies Press.
- [8] Ramakrishnan, Raghu and Gehrke, Johannes (2003), *Database Management Systems*. 3<sup>rd</sup> edition. New York: McGraw-Hill Higher Education.
- [9] FFE Software Inc, *Relational Implementations and Compliance with the Model* (2003), [WWW-document] <http://www.firstsql.com/ireldb3.htm> (Accessed Nov 22, 2006)
- [10] Keith W. Hare, *JCC's SQL Standards Page* (October 26, 2006), [WWW-document] <http://www.jcc.com/sql.htm> (Accessed Nov 20, 2006)
- [11] Douglas, Korry and Douglas, Susan (2006), *PostgreSQL*. 2<sup>nd</sup> edition. Indianapolis: Sams Publishing.

- [12] Jan Wieck, *Slony-I, A replication system for PostgreSQL* (Version 1.0) [PDF-document] <http://developer.postgresql.org/~wieck/slony1/Slony-I-concept.pdf> (Accessed Oct 1, 2006)
  
- [13] Pgpool Global Development Group, *PgPool-II* (Sep 8, 2006), [WWW-document] <http://pgpool.projects.postgresql.org/pgpool-II/en/> (Accessed Dec 20, 2006)
  
- [14] Pgpool Global Development Group, *pgpool-II README* (Jul 23, 2006), [WWW-document] <http://pgfoundry.org/frs/download.php/1083/pgpool-II-1.0.1.tar.gz> (Accessed Jan 1, 2006)